NPS52-89-060

NAVAL POSTGRADUATE SCHOOL

Monterey, California

A Knowledge Based Approach To Program Debugging

Yuh-jeng Lee

September 1989

Approved for public release; distribution is unlimited.

Prepared for:

Naval Postgraduate School
Monterey, California 93943

FedDocs
D 208.14/2
NPS-52-89-060

208. 14/2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. W. West, Jr.
Superintendent

Harrison Shull
Provost

This report was prepared in conjunction with research funded by the Naval Postgraduate School Research Council.

Reproduction of all or part of this report is authorized.

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): NPS52-89-060	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): 52	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Naval Postgraduate School; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: O&MN, Direct Funding
8c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification): A Knowledge Based Approach To Program Debugging

12. PERSONAL AUTHOR(S): Yuh-jeng Lee

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 87/10 TO 89/09	1989 September 13	74

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

We are exploring the use of executable specifications in automating the process of debugging logic programs. We have formulated a computer model that encodes programming knowledge including a classification of program bugs, heuristics that analyze and repair program errors, and operational semantics of the language, and is capable of reasoning with programs and their specifications. The realization of our methodology is the *Constructive Interpreter* which functions as a debugger as well as a program synthesizer. It contains three major components: test case generator, bug locator, and bug corrector. The test case generator can generate test data systematically by executing specifications. The bug locator will automatically locate a bug should the program fail to compute an answer that agrees with the specifications. The bug corrector will analyze the nature of the bug and invoke correction heuristics which might involve the use of (1) a deductive theorem prover which will try to construct a proof and deduce sufficient conditions to amend the program, and (2) an inductive program generator which will synthesize the missing part of the program. In this paper, we present our approach and an implementation of the experimental prototype debugger.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Yuh-jeng Lee	22b. TELEPHONE (Include Area Code): (408) 646-2361; 22c. OFFICE SYMBOL: 52LE

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

1. INTRODUCTION

Ever since the beginning of computer programming, unintentional errors, "bugs," have plagued all but the most trivial software projects. Therefore, a practical, preferably automatic, debugging method or tool would be of great value. However, despite this fact, the issue of software debugging, per se, has seldom been a topic of programmatic research. As compared to other software development activities, such as design, testing, and verification, there is much less research literature or formal methodology that deals with debugging directly.

The goal of this research is to formalize and design a methodology for automating the program debugging process. Utilizing formal specifications and operational semantics, our work is aimed at exploring the possibility of building an automated program debugger that possesses debugging knowledge and reasoning abilities.

Debugging a given program involves three steps: bug discovery, bug location, and bug correction. We focus on the use of executable specifications to generate test cases for bug discovery, locate bugs when test data cause a program to fail, and guide both deductive and inductive bug correction. For programming and specification languages, we use Horn clauses because of their clean syntax, simple semantics, and absence of distracting run-time control details.

In this paper, we will summarize the approach of our work, discuss issues related to program debugging, and present our research results.

1.1. Approach

With the target language being pure Prolog, we have formulated a computer model that encodes domain knowledge necessary for automating the process of debugging Prolog programs. It includes a classification scheme of program bugs, heuristics that analyze and repair program errors, operational semantics of the language, and intended behavior of a program. We have also incorporated in our debugger deductive and inductive inference strategies to reason with programs and their specifications.

The realization of our methodology is the *Constructive Interpreter* which functions as a debugger as well as a program synthesizer. It contains three major components: test case generator, bug locator, and bug corrector. When supplied with a program and its executable specifications, the test case generator can generate test data systematically by executing specifications. The *Constructive Interpreter* then executes the program on the test data. Should the execution fail to return an answer that agrees with the specifications, the bug locator will automatically locate a bug that is causing the failure. The bug corrector then analyzes the nature of the bug and utilizes correction heuristics which guide the use of the specifications and which attempt to repair the bug. This bug fixing process might involve the use of (1) a deductive theorem prover which will try to construct a proof and deduce sufficient conditions to amend the program, and (2) an inductive program generator which will synthesize the missing part of the program.

By applying knowledge of the semantics of a logic programming language, the classification of bugs, and specifications of a program, the *Constructive Interpreter* performs much as an active human expert does during a typical debugging session. When given a program and its specifications, it can (1) execute a goal as a regular interpreter does, (2) generate test cases systematically when symbolic input data are supplied, (3) verify the results of a computation, (4) trace the execution of the program, and (5) locate and fix a bug when a goal does not compute correctly.

The typical process of debugging — designing a test case, detecting an error in the program, locating the error, and fixing it — can also be applied in program synthesis. As a debugger, the *Constructive Interpreter* uses the given program as a starting point in its search for a correct one. As a program synthesizer, it starts the search with an *empty* program. With the application of executable specifications, an inductive search space for programs, and a deductive mechanism for synthesis, this system could be used to allow one to specify a program and give the skeleton of the recursive structure and let the system try to do the rest.

It should be noted that the *Constructive Interpreter* has the capability of executing symbolic data. This is important, since it allows one to use generic test cases. The use of executable specifications is also critical in that it allows the process of checking for errors, in addition to generating test cases, to be automated. To locate bugs automatically, one needs lower-level specifications. That is, to locate an erroneous procedure, one needs the specification for that procedure. Finally, Prolog possesses several advantages over other languages for the purposes of our research: it permits symbolic execution and provides a natural mechanism for writing executable, low-level specifications.

1.2. Specification

A specification is a fundamental concept in engineering. It embraces the idea that one must have a clear notion of the artifact one is going to build before one embarks on building it. The form and context of a specification depend heavily on the nature of the artifact being built.

In software development, the specification may be regarded as an abstraction of the concrete problem at hand, as the starting point for the subsequent program development, and as the criterion for judging the correctness of the final software product. It is a precise and independent description of the expected program behavior, a description of *what* is desired, rather than *how* it is to be achieved or implemented.

Much of research in specification and specification languages is aimed at expressing program requirements in formal, unambiguous, and complete terms. In section 4, we will discuss issues related to software specifications and how we use executable specifications for program debugging.

1.3. Testing, Verification, and Debugging

Within the recently emerged discipline of software engineering, there are techniques that deal with every phase of the software life cycle (e.g., requirement, specification, design, coding, testing, and maintenance) and overall software project management [Boehm-76]. Among these, testing is considered an empirical means of software quality assurance.

Broadly classified, there are two kinds of testing methods: static and dynamic. Static testing performs analyses of programs without executing them at all. Usually, these involve using tools, such as *code inspectors*, to make sure programs adhere to minimum quality standards. Dynamic testing performs analyses of programs by actually executing them on a selected set of input data, sometimes with the help of tools that support the following operations: input setting, stub processing, results display, test coverage measurement, and test planning [Miller-84].

The goal of testing is to provide empirical evidence for the correctness of a program after the testing has been completed. Although a testing run usually only demonstrates some functions of the software, it has been argued in [Goodenough-Gerhart-75] that through a carefully selected data set, one can conclude that the program being tested is correct. The idea is to choose a reliable and valid *data selection criterion* which defines what properties of a program must be exercised to constitute a thorough test. One can then design a *complete* test data set which consists of test cases satisfying all the properties defined by the data selection criterion. If every element of the test data set yields an acceptable result, then the program is correct, since the set is supposedly representative of all the possible legal inputs. Nevertheless, practically, it is quite likely that even programs that have been shown to work correctly for a large set of test runs may not work correctly for a never-used combination of data values.

Another major problem of using testing for software quality assurance is that the testing approach fails to specify how one should proceed in the case where the program behaves incorrectly on some test runs. Generally, the question of how to track down the source of errors and correct it is not dealt with in testing methodology.

Advocates of formal verification methods have suggested that one way to eliminate testing is to provide a correctness proof of the program. This approach is motivated by the fact that it is physically impossible to test most programs over all possible combinations of test data. As has been argued [Dijkstra-76], testing can only show presence of errors, not their absence. Also, one advantage of using formal verification techniques is that they can be applied to many phases of the software life cycle (e.g, specification, design, and coding) and do not require generating test data.

The most common way to prove a program correct is to use invariant assertions. Invariant assertions are statements about conditions and relations between program variables, used as checks against the code to demonstrate program consistency; they must be supplied for each loop. Basically, there is a set of input assertions corresponding to the data properties, and a set of output assertions corresponding to the desired input-output relation. The task is to show (prove) that the output assertions are satisfied after executing a given

program, or that mathematically the program is consistent with its predefined assertions.

In general, nontrivial programs are very complicated and time-consuming to prove, and any proof procedure for correctness is only partially automatable. Some programs for which proofs have been completed and published have later been found to contain easily detectable errors (e.g., see [Goodenough-Gerhart-75]).

Still, the major problem is similar to that of testing: What if a program cannot be proved correct? In other words, one cannot dispense with debugging, no matter how good he is at proving programs, since one will not be able to prove an incorrect program.

1.4. Symbolic Execution

With symbolic execution, instead of supplying specific constants as input values to a program being tested, one supplies symbols. Symbolic execution systems (e.g., [Clarke-76, King-76, Howden-77, Clarke-Richardson-81, Cohen-Swartout-Balzer-82, Howden-86]) allow input variables in a program to take on symbolic as well as numeric values. A symbolic vaule can be an elementary symbolic value or an expression in numbers, arithmetic operators, and other symbolic values. It represents some *unknown* yet *fixed* value. The normal computational definitions for the program can then be expanded to accept symbolic inputs and produce symbolic formulae as output which can be checked, manually or automatically, to verify the correctness of the program being symbolically executed.

The main advantage of symbolic testing is that performing a single *symbolic execution* is equivalent to a large, possibly infinite, number of normal test runs. By varying the degree to which symbolic data is introduced into the symbolic execution one can, in some cases, provide a proof of correctness.

The whole problem with symbolic execution is that there is no way to determine which way a test is to go. Such a system, therefore, usually resorts to asking the user, or to backtracking over different choices. In Prolog, backtracking is automatic whenever a goal fails.

1.5. Expertise

An expert system is a computer program that implements the skills of human experts and aims at providing expert level solutions to problems of interest. Given the fact that debugging is one of the most frequently engaged activities and that there is expertise involved, an expert debugging system is not only feasible, but also valuable.

In the domain of program debugging, a human expert debugger's behavior can be classified into two main categories. For a majority of bugs manifested by compiler error messages or run-time "illness" (such as dividing by zero, or taking the square root of a negative number), a person with a highly trained question-answer skill or with abundant debugging experience can usually find the cause of the bug almost "off the top of his head."

On the other hand, in a faulty program there might be subtle logical errors which are "symptomless," and these cannot be detected with the expert's top-of-the-head skill. Usually, this kind of error can be resolved only after careful reasoning about the performance of the program or experiments.

Therefore, an expert debugging system should be ideally constructed in a way that incorporates both of these kinds of capabilities: a quick symptom-diagnosis skill, and a reasoning capability based on the understanding of the program. These two components have been termed the *shallow* and the *deep* models of debugging, respectively [Badger, *et.al.*-82].

The shallow model implements the domain specialist's top-of-the-head-skill. Usually, this needs judgemental heuristics acquired from the domain experts (and, therefore, the shallow model is also called a "heuristic model"). This knowledge is then coded in the "**if** *condition* **then** *action*" production-rule form, and the inference process performs backward or forward reasoning through production rules by creating associations between conditions/facts and actions/conclusions. Although this is a "surface knowledge approach" (i.e., the system does not need to understand the problem it is solving, since its intelligence comes from pre-coded human reasoning), and it suffers from the disadvantage of not being likely to cover all cases exhaustively, the heuristic model of debugging does have one major advantage: third, since the system's reasoning is straightforward (i.e., once a condition invokes a rule, a certain action will take place), it can be fine tuned to achieve impressive performance. An expert debugging system of this nature has been successfully implemented in a Pascal programming environment [Harandi-83]. This system embodies heuristics that deal with most of the compile time and certain run time errors with sufficiently explicit symptoms.

The deep model implements the domain expert's logical and causal structure of domain knowledge, as his understanding and reasoning abilities. For program debugging, an ideal expert system of this kind should be able to accept a buggy program and a nonalgorithmic description of the program's intended behavior, perform necessary analyses, locate bugs that cause the program to behave incorrectly, and suggest corrections — all with little user involvement. Better yet, the system should have the ability to test its own solutions and make further revisions on its previous suggestions until the program behaves correctly. Unquestionably, this system has to have a large amount of knowledge that has to do with the semantics of programming languages, types of errors, causes of errors, means of correcting errors, and domain-dependent facts concerning the problem that the program is supposed to solve. In addition, it has to have effective reasoning strategies that utilize the knowledge and analyze the program's behavior.

1.6. Program Synthesis

A major use of software specification is to provide a very high level descriptive tool so one can build a large system in top-down fashion. If the specification truly embodies what one needs, then one should be able to provide that abstract specification as input to an automatic programming system and be able to receive, as output, a low level program that

can be executed on the target machine more efficiently. Ideally, this practice would salvage much of the grievance in software development processes. Given the current state of technology, however, such an automatic programming system is still not available for general software production.

By restricting the problem domain of such a system, it is possible to apply such technology and build systems for practical applications. The system described in [Barstow, *et.al.*-82] deals with a class of numerical software for scientific processing and has allowed the client scientists both greater flexibility in their ability to specify program behavior and much more rapid program development to establish the validity of that behavior.

Research in logic programming could also contribute in this area, since the development of logic programs from specifications should be simpler than that of conventional programs. One can then focus on the mechanisms and strategies of program synthesis. We discuss this issue in section 8.

1.7. Overview of the Paper

Our work focuses on the investigation and implementation of an automatic debugger that possesses deep level reasoning abilities. Section 2 reviews related work in this area. The target and implementation language for the system developed in this paper is Prolog, a logic-based language. Section 3 presents a brief introduction to the basics of Prolog programming and discusses the concept of meta-programming which is the foundation for building the inference engine of our debugging system. Section 4 explains the idea of executable specification and its role in our automatic debugging enironment. Utilizing executable specifications, an interpreter can check and monitor the results of program execution. Section 5 deals with such an interpreter that can locate bugs automatically. Bug correction requires a large amount of knowledge which must be formalized in an automated environment. Section 6 analyzes bug correction heuristics in Prolog, which include deductive and inductive code generation. Section 7 discusses details of the *Constructive Interpreter*, a Prolog implementation of our debugging methodology, and demonstrates its behavior through several examples. With the availability of the *Constructive Interpreter* and executable specifications, the logical next step is to synthesize programs from specifications. Therefore, section 8 is on automatic program synthesis. Lastly, section 9 gives the concluding remarks and directions for future research.

2. APPROACHES TO AUTOMATED PROGRAM DEBUGGING

When a programmer debugs a program, he has to know the problem that the program is supposed to solve, how the program is executed, how to track down errors if the program does not behave as expected, what the likely causes of errors are, and how to fix errors if they occur in the program. For a computer system to debug a program automatically, it is evident that it requires (or must access to) the same knowledge. Automated debugging systems can, therefore, be analyzed according to the following characteristics: (1) how the intended behavior of a program is specified and supplied to the debugger; (2) how the errors

in a program are discovered; and (3) how the errors are located and corrected.

We will review some of the more noteworthy approaches to automated debugging in the following sections and analyze their features.

2.1. Logical Analysis of Invariant Assertions

From the viewpoint of program verification, Katz and Manna [Katz-Manna-75, Katz-Manna-76] have demonstrated how invariant assertions can be used for automatic diagnosis and correction of logical errors. If a program can be verified as correct, there will be no need to debug it. However, failure to prove correctness of a program might be because the program is actually incorrect, or the program is really correct but we are unable to prove it. In either case, one won't be able to draw conclusions about how the program is meant to behave. Therefore it is necessary to discover how the program has to be modified to meet its original specifications.

Consequently, there are two different approaches for which invariants can be used to debug a program. First, prove that the program is incorrect. That is, show that for some legal input, either the program does not terminate, or terminates with a wrong result. Then use the proof with the aim of helping locate and eliminate errors. This is possible because any change in the program, when it is intended to correct errors, must change at least one of the invariants used in the incorrectness proof. Thus, corrections can be restricted to those which will influence invariants from a proof of incorrectness. However, sometimes the proof is not very useful for guiding the search for corrections.

The second approach is to fix the program so that a proof of correctness is guaranteed to succeed, without necessarily showing that the original program is incorrect. In this case, one needs user-supplied or machine-generated invariant assertions for each loop. Code is then synthesized to meet invariants that suffice for a correctness proof. A similar, but more structured, approach can be found in [Dershowitz-83].

The purpose of this approach is that by modifying the invariant assertions to derive the desired output assertions, one can, hopefully, fix the bugs in the original program. However, even though the assertions reflect relationships among program variables and could be used to guide the correction of bugs, intrinsically there are difficulties with this method.

First, it is very difficult to give complete sets of assertions that describe the problem entirely, because the assertions are *static descriptions* of a program's task. They only provide information about *what* the relations between variables should be, not *how* the relations between variables can be established. Therefore, one can easily generate programs that satisfy the intended output assertions without performing the task in the required manner. This problem holds for any method based on nonalgorithmic specifications.

Second, it is often the case that formulating invariant assertions to prove the correctness of a program *is more difficult* than writing a correct program itself. Even with

the help of some automatic invariant generators and theorem provers, the general problem of finding an algorithm to generate invariant assertions for any program is unsolvable, just as showing a program to be correct is.

2.2. The Intelligent Program Analyzer

Based on the premise that a system cannot satisfactorily analyze a program unless it can "understand" what it is examining, Ruth has constructed a prototype intelligent program analyzer [Ruth-76] which, when accompanied by a task description and directed by the structure of the program to be analyzed, functions like a programming expert and is able to comprehend, verify, and, if necessary, correct a given program.

In this system, the most important thing is to represent an algorithm in a form that the program analyzer can utilize. This representation, termed "program generation model" (PGM), is expressed in terms of some universal constructs and mechanisms such as loops, conditionals, flags, calculation of expressions, etc. In running the system, both the algorithm supplied by the user and the given program to be analyzed are organized as lists of *actions*, which are represented by the algorithm description primitives such as

(LOOP parameter–list ACTION–list),
(TEST predicate ACTION's–if–true ACTION's–if–false),
(ASSIGN var expr), and
*(INTERCHANGE var*1 *var*2), etc.

This is because the top-level driver for the analyzer is basically an *action list matcher* (ALM). The analyzer then uses two lists of actions and runs a top-down analysis. It proceeds with attempts to match the *actions* of the PGM and the given program. If an *action match* succeeds, the ALM will try to match the next *actions* on the *action lists* with the parameters given for those *actions*. Matching continues in this incremental fashion until there is a "non-recoverable" difference or the *action lists* have been exhausted. In the former case, the analyzer identifies an important difference between the PGM and the given program, which might be the sign that there is a bug in the program (assuming the specification given to the PGM is correct).

The algorithm needed for solving the problem provides a *dynamic description* of the program's task. For the purpose of debugging this approach seems to be more useful than deriving invariant assertions for a program. Since in this case we are giving information about *how the program must proceed* instead of *what it has to achieve*. In other words the debugger has knowledge of how to complete the program's goals rather than just the goals themselves.

Although analyzing the computation process by comparing a program with its intended algorithm specifications might sound promising, there are certain limitations on this

approach. First of all, there is the problem of translating both the algorithm and the program into comparable forms. It is often the case that an algorithm specified in semi- or non-procedural form is simpler, in terms of structure complexity, than a complete program. Since algorithms and programs are usually at different levels of abstraction, some information might be lost or misrepresented in the transformation process.

Second, supplying the whole algorithm in sufficient detail for the machine to execute might become cumbersome, especially when the problem to be solved is rather complex.

2.3. The LAURA System

In an attempt to deal with the problem of automatic checking for the equivalence of two programs, LAURA [Adam-Laurent-80] was built with the purpose to automatically debug student programs. This system differs from Ruth's program analyzer in that LAURA emphasizes the *source-to-graph transformation* and considers debugging from the perspective of the *comparison of two calculus processes*.

Basically, in LAURA, a graph is built up from a program or an algorithm description. That is, a graph is a representation of the calculus process implied by the program or algorithm. In a graph, the nodes represent the various operations of the calculus process (assignments, tests, inputs, outputs) and the arcs represent the flow-graph defined on these operations.

In a debugging session, both the intended algorithm and the program to be debugged are first transformed into their equivalent graphs. The LAURA system then compares a region of one graph with any region of the other. It is therefore possible to recognize that a certain part of one program computes the same functions as a certain part of the other. In other words, parts of a program can be identified. If a total identification of the given program is possible, then it can be declared correct. Otherwise, the unidentified parts might contain errors. At the end, the system is able to make several diagnoses of errors if there are unidentified regions on the graph representing the program.

In the LAURA system, both the algorithm and the program are transformed into graphs. This is advantageous in that the analysis can be performed independently of the language in which the program is given. However, this system has the other drawbacks discussed in the previous section on the intelligent program analyzer: the problem of transformation and the need for the user to supply a complicated algorithm.

2.4. Algorithmic Program Debugging

Applying the diagnostic algorithms in a program synthesis system, Shapiro [Shapiro-83] has designed an interactive debugging system for Prolog. The system does not require the programmer to provide either an intended algorithm or a specification. Rather, the system simulates the execution of the target program on a given input, and generates results for each procedure call. The programmer is then queried for the correctness of some

intermediate results of procedure calls. Based on the answers to the queries, the system can diagnose an error by isolating an erroneous procedure, and suggests a correction (for a limited class of errors). The same approach was applied to Pascal programs in [Renner-84].

The bug locating algorithm could be very useful for debugging logical errors, especially for predicate-oriented programming languages. However, due to the fact that it requires the user to act as an oracle and to verify the correctness of results from executing procedures, this algorithm has some limitations. When a procedure contains a complex task of computation, the user might not be able to verify an intermediate result without additional help. Also, the user has to be fairly involved in answering queries generated by the system. To improve the efficiency, Plaisted [Plaisted-84] suggests a more efficient algorithm to diagnose an incorrect clause. It has been shown that this improvement is optimal to within a constant factor for space, time, and number of queries to the user. Using information about dependencies between subgoals (such information can be obtained by using a suitably modified unification algorithm as described in [Bruynooghe-Pereira-84]), Pereira [Pereira-86] has designed a method called *rational debugging* the object of which is to discover discrepancies between a program's purported models and its actual input/output behavior, as manifested by running the program.

2.5. PROUST: Knowledge-Based Program Understanding

PROUST [Johnson-Soloway-85], a knowledge-based program understanding system which does online analysis and understanding of Pascal programs, is intended to be a tutoring system to assist novice programmers in learning how to program. It has a knowledge base of programming plans and strategies, together with common bugs associated with them. Given a program and a nonalgorithmic description of the program requirements, PROUST tries to find the most likely mapping between them by reconstructing the design and implementation steps that the programmer has gone through when writing the program. This reconstruction process is meant to recognize the *plans* used by the programmer. It is also the basis to program understanding.

Program bugs are detected when there are mismatches between the reconstructed *plans* and code. Some mismatches are not interpreted as bugs; they are simply recognized, using *plan transformation rules*, as common implementation variants of expected code. Others can be recognized as instances of specific program bugs by *bug rules*. Other than these two cases, mismatches that cannot be accounted for might result in incomplete or aborted program analysis; or they might be interpreted as bugs, might be ignored, or might bring out warning messages.

2.6. Talus: An Intelligent Tutoring System

Talus [Murray-86] is an automatic program debugging system that both detects and corrects bugs in student programs written to solve small but nontrivial tasks in pure Lisp. It does not rely on the programmer's assistance. Debugging with Talus consists of three stages: algorithm recognition, bug detection, and bug correction.

Representations of algorithms are stored and heuristics are used to match the program to be debugged with the algorithm most similar to it. For this purpose, all functions in the student programs and the stored algorithms are parsed into *E-frames* which represent abstract properties of recursive functions that enumerate the elements of a recursively defined data structure. Talus performs a best first search to choose between competing algorithms and to map student functions to stored *E-frames*. The matching process include *algorithm identification*, *function mapping*, and *formal variable mappings*. Bug detection is done by constructing an inductive proof that student programs are equivalent to the stored functions. A bug is present when the equivalence relation cannot be established. Heuristic methods are then used to suggest alterations, by consulting the stored algorithms, to the student's functions to remove bugs.

2.7. Summary

Based on the characteristics listed in the beginning of this section, we now summarize the above research on automated debugging. To specify the intended behavior of the program to be debugged, the majority of researchers have chosen to use fairly intensive, complete descriptions of the algorithm. They either built *model programs* (such as [Ruth-76, Adam-Laurent-80, Murray-86]) or require *program descriptions* (such as [Johnson-Soloway-85]). This approach seems feasible in building tutoring systems, since in this case, one can use a single *model program* to debug many student programs which, supposedly, are aimed at solving the same problem. Using invariant assertions to debug programs requires input/output specifications written in predicate calculus [Katz-Manna-75]. The system in [Shapiro-83] does not need a pre-stored algorithm or specification, however, the user is required to supply information on expected program traces.

Those systems that use model algorithms (including [Ruth-76, Adam-Laurent-80, Johnson-Soloway-85, Murray-86]), predictably, have to rely on some kind of matching between algorithms and programs to detect and locate errors. A mismatch usually signals the existence and points to the location of a bug. In [Katz-Manna-75, Murray-86], formal verification techniques are used. Bugs are detected when the effort to verify the program specification fails, and the part of the program that causes the failure usually contains the bug. For the system in [Shapiro-83], the difference between the expected and actual execution traces is used to locate the bug, however, the user has to supply the initial test case that shows the program is buggy.

Correcting errors is not treated in [Ruth-76, Adam-Laurent-80, Johnson-Soloway-85]. All the others rely on heuristics of one kind or another. The strategy in [Murray-86] is to use the stored functions to modify the program so that a successful verification can be obtained, while in [Shapiro-83] a new part of the program is synthesized.

Comparing our *Constructive Interpreter*, we use executable input/output specifications to define the intended behavior of a program and to generate test cases for bug discovery. We employ the execution mechanism of a Prolog machine to locate bugs, using specifications to validate computation results. We also have heuristics to analyze bugs and suggest fixes, and use techniques in deductive theorem proving and inductive synthesis to

mechanize the bug correction process.

3. LOGIC PROGRAMMING

Logic is the study that is concerned with relationships and implications between assumptions and conclusions. It has long been used as a means for formalizing natural language and human reasoning. *Logic programming* uses *logic* to express information and present problems to a computer, and uses *logical inferences* to solve these problems. It has become one of the most rapidly growing areas of programming language research.

Broadly defined, a logic programming language is a language that is based on a formal logic system, with operational semantics defined by deduction in that system. Lisp (or more precisely, pure Lisp), for example, is a logic programming language based on the λ–calculus. Languages based on equational logic, such as EQLOG [Goguen-Meseguer-84] and rewrite systems (e.g., [Dershowitz-84]), also fall into this category.

A narrower definition of logic programming refers to the use of first-order predicate logic, or a subset of it, as a programming language, with emphasis on using predicates and deduction to describe computation. Based on the resolution principle [Robinson-65] and its successive improvements, efficient schemes for processing predicate logic have been recently developed. The principal idea is to represent programs with the Horn clause (also called definite clause) subset of the first-order predicate logic [Kowalski-74, Kowalski-Emden-76]. This breakthrough set the basis for procedural interpretation to Horn clause logic and accelerated progress in the development of logic programming languages. Prolog [Clocksin-Mellish-84], one of the most recognized logic programming languages, is nowadays a viable alternative to Lisp in symbolic processing and artificial intelligence research.

In the following sections, we present a short introduction to pure Prolog, the target language for the debugging system developed in this paper, and discuss related issues including our rationale for choosing Prolog as a tool for our research.

3.1. Basics of Prolog Programming

The central idea of Prolog programming is that a program is a sequence of logic statements of a restricted form — Horn clauses, and the execution of such a program is a controlled logical deduction from the clauses forming the program.

A Horn clause is either a simple atomic *assertion* or an *implication*. Every assertion is a simple term written as

$$p$$

whereas every implication is of the form

$$p \leftarrow p_1, p_2, \cdots, p_n$$

where the conclusion p is called the *head* of the clause, and the conditions $p_1, \cdots, p_n$ are the *body* of the clause. The head p and the p_i's are all logical atoms (predicates). Both of these

two kinds of clauses are universally quantified logical sentences.

The declarative meaning of a Prolog program can be viewed as a collection of assertions and implications about some relations, i.e., the conjunction of the universally quantified Horn clauses in the program. The assertion p means "p is true" and the implication $p \leftarrow p_1, \cdots, p_n$ means "p is true, if $p_1, \cdots, p_n$ are all true."

Computationally, a Prolog program consists of a number of procedure definitions. Clauses with the same functor form a procedure, with different clauses in a procedure represent alternative cases of that procedure. Interpretated procedurally, the clause p is read "goal p is satisfied", and the clause of implication is read "to satisfy goal p, satisfy goals $p_1, \cdots, p_n$."

Using the procedural interpretation, a set of definite clauses P can be executed as a logic program by giving a goal, or a conjunction of goals. Ideally, the computation progresses via nondeterministic goal reduction. For example, to compute some initial goal (or subgoal) p, a clause

$$p' \leftarrow p_1, p_2, \cdots, p_n$$

in P is chosen nondeterministically aiming at unifying p with p'. The unification succeeds if the goal and the clause head can be made identical by "filling in" suitable values for the variables, renames, if necessary to make them distinct. If this is the case (a match), then the original goal can be reduced to

$$(p_1, p_2, \cdots, p_n)\theta$$

($p_i\theta$ denotes the result of applying the unifying substitution to each occurrence of the unified variable in p_i). The computation continues with the aim of satisfying the reduced goal in any order, and terminates when the current goal becomes empty. If a computation succeeds, the final values of the variables in p are conceived of as the output of the computation. Because of the nondeterministic nature of this process, a given goal can lead to a number of successful and unsuccessful computations. Variable bindings from a successful computation can be conceived as output.

The operational semantics of Prolog is based on SLD-resolution (Linear resolution with Selection function for Definite clauses, see [Apt-van Emden-82]). That is, Prolog's execution follows a sequential simulation of the nondeterministic computation, using a depth-first search strategy with a backtracking mechanism incorporated. In the computation process, Prolog will try all unifiable clauses sequentially, in the order they occur in the program text, and subgoals are solved from left to right. When it fails to find a clause whose head can be unified with the current goal, it backtracks to the most recently executed goal, undoes any substitutions made by the unification, and tries to resatisfy that goal with a different solution. If none can be found, the entire computation fails.

The computation process can also be described as the traversal of a computation tree. A computation tree T of a program P is a rooted, ordered tree. Each node in the tree has the form $p(x, y)$, where p is a procedure (predicate) name, x and y represent input and output vectors over some domain. For the clause

$$p(x, y) \text{ :- } p_1(x_1, y_1), p_2(x_2, y_2), \cdots, p_k(x_k, y_k)$$

involved in a computation, the corresponding part in T includes the internal node $p(x, y)$ and its sons $p_1(x_1, y_1)$, $p_2(x_2, y_2)$, $\cdots$, and $p_k(x_k, y_k)$. In other words, all the p_i's are procedures invoked from within p, in that order. The interpretation is as follows: procedure p, on input x, calls p_1 on x_1, and if this call returns y_1, then p calls p_2 on x_2, and if this call returns y_2 then $\cdots$, then p calls p_k on x_k, and if this call returns y_k, then p returns y as its output. If a node $p(x, y)$ has no sons, it means that the procedure p, on input x, has a legal computation that returns y without performing any procedure calls.

Programs, for our purposes, are presumed to obey their Horn clause declarative semantics, i.e., "extra-logical" features, such as cuts, clause order, and subgoal order, may affect efficiency and termination, but not correctness.

3.2. Why Prolog

The activities of debugging, maintaining, and extending programs are largely dependent on reading and understanding a program as it is represented in a programming language. Also, it has been observed that professional programmers spend more time examining existing programs than they spend writing new programs. Therefore, the characteristics (or design) of programming languages does have a significant bearing on software errors.

Although it is debatable whether a general-purpose programming language is feasible, it is obvious that a language that is designed for a wide range of applications and users must have conflicting objectives such as ease of learning, ease of use, ease of extending the language, ease of extending programs written in the language, resistance to programming errors, machine independence, efficient object code, and efficient compilation, etc. [Myers-76]. In other words, if a language attempts to satisfy all of these goals, it will no doubt be less than ideal for each particular type of application or audience.

For the purpose of our research, we would like to design a system that can understand programs, possesses knowledge about the programming language, and has a reasoning power that utilizes its knowledge to analyze programs. Thus, the essential features of the language, in our case, are readability of programs written in the language, the ability to represent clearly and simply the program's function and logic, and well-understood declarative and operational semantics. Since we are to investigate effective strategies of debugging "difficult" errors, we want to be able to concentrate on the logic (meaning) part of the program without worrying about the control (efficiency) problem (cf., [Kowalski-79]).

Formal logic is one of the most widely used schemes for knowledge representation, and its deductive system is guaranteed correct to an extent that other representation schemes have not yet reached. Beside, derivations of new facts can be mechanized. Prolog is renowned for its simple syntax and well-defined semantics. Pure Prolog is also lacking those run-time control mechanisms that usually complicate the analysis of a program's logical properties. Other features, such as meta-programmability and executable specification, will

be discussed in detail in the following section and the next section.

3.3. Meta-Programming

One important feature of Prolog is that it can easily be used to build a system that manipulates and executes other programs written in that language. As pointed out in [Fuchi-Furukawa-86], meta-programming can be characterized as the ability: (1) to handle a program as data; (2) to handle data as programs and to evaluate them; and (3) to handle a result (success or fail) of computation as data.

This meta-programming capability is essential when implementing a system to reason about programs. It provides a basis for building a powerful programming environment. Prolog is especially attractive in this aspect, since one can easily write a meta-interpreter to execute pure Prolog programs in just three statements:

A Prolog Interpreter		
interpret ((*G* 1,*G* 2))	:-	*interpret* (*G* 1), *interpret* (*G* 2)
interpret (*Goal*)	:-	**system**(*Goal*), *Goal*
interpret (*Goal*)	:-	**clause**(*Goal* ,*SubGoals*), *interpret* (*SubGoals*)

The first clause solves a conjunctive goal by recursively solving its components. The second clause checks if (a noncomposite) *Goal* is a system (built-in) predicate (**system** itself is a built-in predicate that succeeds if *Goal* is a call to a built-in procedure) and, if it is, executes the goal directly. The third clause uses a built-in predicate **clause** both to find a clause whose head can be unified with *Goal* and to reduce *Goal* to the list of subgoals in the body of that clause. The interpreter then solves these subgoals recursively. As will be seen, our debugging system, for Prolog programs, is based on the scheme of this interpreter.

4. EXECUTABLE SPECIFICATIONS

Since a software specification serves as a blueprint for the program to be built, it must be clearly and unambiguously understandable by the software specifier and the implementor. For this reason, we argue that the specification must be constructed in a formal language (more in next section).

A specification must be testable: it must be possible to ascertain whether an implementation has fulfilled the requirements of the specification (i.e., whether the implementation is consistent with the specification). For this purpose, a specification must be complete and formal enough that it can be used to determine if a proposed implementation satisfies the specification for arbitrarily chosen test cases. That is, given the

results of an implementation on some arbitrarily chosen set of data, it must be possible to use the specification to validate those results. This implies that the specification can act as a generator of possible behaviors which must be present in the proposed implementation.

On the other hand, a specification must be tolerant of incompleteness and augmentable. This is because a specification is always a model — and abstraction — of some real or envisioned condition. Furthermore, as it is being formulated it will exist at many levels of detail. The analysis tools employed to aid specifiers and to test specifications must be capable of dealing with incompleteness, and the specification must be in a form such that it can be easily modified (i.e., maintained).

Last, but not least, a specification must be localized and loosely coupled, and the specification language must provide facilities for modular construction of specifications. This requirement arises from the dynamic nature of specification. Although the main purpose of a specification is to serve as the basis for design and implementation of some system, it is not a precomposed static object, but a dynamic one which undergoes considerable modification. Such modification could occur in three main activities: (1) formulation, when an initial specification is being created; (2) development, when the specification is elaborated during the iterative process of design and implementation; and (3) maintenance, when the specification is changed to reflect a modified environment or additional functional requirements. With so much change occurring to the specification, it is critical that its content and structure be chosen to accommodate this activity. The main requirements for such accommodations are that information within the specification must be localized so that only a single piece (ideally) need be modified when information changes, and that the specification is loosely structured so that pieces can be added or removed easily, and the structure automatically readjusted.

4.1. Formal vs. Informal Specification

As one might expect, the formal rigor of a specification varies widely. At one end, research in formal specification languages has become a major area in computer science (cf., [Spitzen,*etal.*-77, Guttag-Horning-78, Liskov-Berzins-86]). A specification is formal if it is written entirely in a language with an explicitly and precisely defined syntax and semantics. At the informal end of the spectrum, the specifications can be expressed in some convenient combination of English, diagrams and a variety of mathematical notations (or whatever the specifier likes).

There are many advantages for using a formal specification. Because of the mathematical nature, a formal specification encourages rigorous descriptions when defining problems. With the well-defined and unambiguous semantics of the specification language, there is only one way to interpret a formal specification, thus reducing the likelihood of misinterpretation and confusion. A formal specification can also be processed by a computer more easily, and, therefore, certain forms of inconsistency or incompleteness in the specification maybe detected automatically.

On the other hand, informal specifications cannot be studied mathematically. It is

easy to hide incompletely designed program behavior under vague informal descriptions. Informal specifications appear easier to construct because they are usually incomplete and imprecise. However, it is very difficult to have them processed mechanically. The main problem for an incomplete specification is the inability to test correctness of an input/output pair.

In automating the software development process, the use of formal specifications is one of the key issues.

4.2. Executable Specifications and Rapid Prototyping

As software projects get more and more complicated, the need for rapidly available prototypes becomes more essential. Research in executable specification languages is gaining popularity partly due to the ever increasing interest in the *rapid prototyping* of software, with the hope that such a prototype, an incomplete or partial functional model of the target system, might provide more information than any formal document.

If the specification is formulated in a language that has operational semantics, the specification can be regarded as a prototype, and its behavior may be scrutinized to determine if it is in fact the behavior of the desired software product. According to [Smoliar-82], there are four major approaches to executable specifications: lambda calculus [McCarthy-60], functional expressions [Backus-78], data flow constructs [McGraw-80], and algebraic representations [Goguen-Tardo-79].

Based on the desired properties for an executable specification language, however, a logic-based language would serve the purpose as well (cf., [Clark-81, Kowalski-85, Komorowski-Maluszynski-86]), if not better, since it is a formal language, and has simple syntax, well-defined declarative semantics, and a well-understood deductive mechanism. Being simple in syntax makes the specification easier to understand. Having a well-defined declarative semantics facilitates the construction of high-level specifications, since a specification language is to describe intended behavior (i.e., *what*) without prescribing a particular algorithm (i.e., *how*). The deductive mechanism provides operational validation of the specifier's intentions. Besides, logical formulae can be written incrementally and can easily be modified and augmented. All these features facilitate the testing and validation of the prototype.

4.3. Specifications in Prolog

First-order predicate calculus has long been used as a specification language. The typical approach to program verification (e.g., [Hoare-69, Katz-Manna-76]) expresses specifications in first-order logic, and relates them to conventional programs by defining the semantics of programs in a "programming" logic. In contrast to this complicated situation of the need to deal with two different languages when verifying the properties of programs, logic programming offers an attractive alternative, namely, the ability to use logic for both specification and computation in a natural way. In addition to the simple syntax and well-

understood semantics, this advantage is rarely met by other traditional programming languages.

Since Horn clauses are a powerful subset of first-order logic, Prolog can often be used for specification with the advantageous extra feature of executability: a program's specifications can be written in Prolog itself and can be executed by the Prolog interpreter or compiler directly.

For our debugging purpose, the specifications of a program describe the relationships between program variables by giving input/output constraints. They define the functionalities of the program without imposing the restriction on how these functionalities are to be achieved. The specifications can be viewed as procedural abstractions (cf., [Liskov-Berzins-86]). A procedural abstraction performs a mapping from a set of input values to a set of output values. Its use was motivated largely by a desire to prove that a program has certain properties (e.g., correctness). Usually, the input/output relation is expressed as assertions which are attached to a program, and the task is to prove that the assertions are true whenever control reached the corresponding points in the program. Each pair of assertions acts as a specification for the program fragment between them. A procedural abstraction can also be obtained by giving an operational specification which explicitly represents an algorithm (or program) that transfers the legal input state to a legal output state.

It may then be argued that the specifications are no different from programs. Indeed, in logic programming, as [Kowalski-85] has contended, *execution efficiency* is the main criterion distinguishing between programs and complete specifications. The specification emphasizes clarity and simplicity but not efficiency, while in the implementation of programs, efficiency is the main consideration. In other words, specifications written in Prolog can be conceived of as *nonalgorithmic*, *executable*, and perhaps *inefficient* programs. In section 8, we will discuss the issue of using specifications to automatically generate programs.

High among the desirable properties of most programs is termination. One way of specifying how progress is being made is to provide a well-founded ordering of input arguments for recursive procedures. A well-founded ordering > is a binary relation on elements of a nonempty set S such that the relation is transitive, asymmetric, and irreflexive, and such that S has no infinite descending sequences. The ordering specifies, for a particular recursive call, which arguments should be decreasing. This is used for detecting looping.

Although more expressive languages, e.g., EQLOG [Goguen-Meseguer-84], HOPE with unification [Darlington,*etal*-85], and RITE [Dershowitz-Josephson-86] may be more suitable for specification, for our research, specifications written in Prolog is readily executable by a Prolog compiler or interpreter.

4.4. Generation of Test Cases

As mentioned earlier, running the specifications of a program not only can test the desired output, but can also generate useful test cases for that program, provided that axioms for primitive predicates are supplied. The information contained in a specification regarding the expected output behavior is indispensable for checking the correctness of the results of program execution, while test cases help reveal instances of incorrect output.

To generate test cases for a given goal, we first run the specifications of that goal to obtain a pair consisting of an input along with its expected output. (In this research, we presume that the specifications faithfully reflect the intended requirements of the program (cf., [Gerhart-Yelowitz-76]) — debugging of specifications is an important research topic, however, we will not get into it here.) We then use only the input value to run the goal on the program to be debugged. If the execution fails, goes into a loop, or returns an incorrect output value, then this test case has showed us that there is at least one bug in the program. If one of the predicates in the specification of a program is defined in the form of a "generator," then utilizing Prolog's built-in backtracking facility, we can generate alternate test cases. If we use a breadth-first mechanism to generate test cases, we can generate a complete (perhaps infinite) set of test cases for that program.

For example, suppose we have the following specification for a sorting procedure:

```
spec(sort(Input,Output))   :-   ordered(Output),
                                 perm(Input,Output)
```

which says that feeding a list "*Input*" to the procedure *sort*, the list "*Output*" is a correct result if it is in order and is a permutation of "*Input*". Suppose *perm* is defined in the following way that generates all possible permutations of a list:

```
perm([],[])
perm([X|Xs],Ys)   :-   del(X,Ys,Zs),
                       perm(Xs,Zs)

del(X,[X|Xs],Xs)
del(X,[Y|Xs],[Y|Ys])   :-   del(X,Xs,Ys)
```

and *ordered* is written as

```
ordered([])
ordered([X])
ordered([X, Y|Xs])   :-  lt(X, Y),
                         ordered([Y|Xs])
ordered([X|Xs])      :-  lt_all(X,Xs),
                         ordered(Xs)
```

where [ ] denotes the empty list in Prolog, and $[X|Xs]$ denotes the list with head X and tail Xs. If we have available the following basic axioms for the primitive predicate *lt* (which means *less than*) as

```
lt(X, Y)        :-  is_number(X),
                    is_number(Y),
                    X < Y

is_number(0)
is_number(X)    :-  is_number(Y),
                    X is Y+1
```

then by executing *spec*(*sort*(*Input*,*Output*)) with uninstantiated variables we can obtain a sequence of input/output pairs, as demonstrated by the following example (for the examples used in this paper, user input is shown in bold face):

```
| ?- spec(isort(Input ,Output)).

Input  = []
Output = [] ;

Input  = [X]
Output = [X] ;

Input  = [0,1]
Output = [0,1] ;

Input  = [1,0]
Output = [0,1] ;

Input  = [0,1,2]
Output = [0,1,2] ;

Input  = [0,2,1]
Output = [0,1,2] ;
```

```
Input  = [1,0,2]
Output = [0,1,2] ;

Input  = [1,2,0]
Output = [0,1,2] ;

Input  = [2,0,1]
Output = [0,1,2] ;

Input  = [2,1,0]
Output = [0,1,2]

yes
```

As can be seen, *spec*(*sort*) first generates an empty list as input, then a one-element list, then two-element lists with all possible permutations, then three-element lists with all possible permutations, etc. The variable *Output* shows the expected result for each given input, and can be used to verify the correctness of a program (see next section). Note that the one element list is symbolic, hence an infinitude of test cases are captured in one fell swoop.

4.5. Validation of Computation Results

When a program is to be debugged, we assume that the properties of each procedure in the program can be described by the program's specifications. These nonalgorithmic specifications detail the relationships between program variables as well as the well-founded ordering under which successive input values to recursive procedures form a descending sequence. In other words, they define all legal input/output pairs for each procedure. Unspecified procedures are presumed correct (note that this is an example of handling incomplete specifications).

Suppose we have a relation R that is defined by specifications S and is to be computed by program P. If every instance of R computed by P can also be deduced from S, then P is *partially correct* with respect to S, i.e.,

$$\textbf{if } P \vdash R \textbf{ then } S \vdash R,$$

where $X \vdash Y$ denotes that conclusion Y can be derived or proved from assumption X. This actually means that the program P is consistent with the specification S, or

$$S \vdash P.$$

If there is a computation result of P that cannot be deduced from S, then P is *incorrect* with respect to S.

On the other hand, if every instance of R defined by S can be obtained by executing P, then P is *complete* with respect to S, i.e.,

$$\textbf{if } S \vdash R \textbf{ then } P \vdash R.$$

This means that the program P derives every instance of R that is defined by the

specification S, or

$$P \vdash S.$$

If there is an instance of R that is defined by S but cannot be the result of executing P, then that instance is "uncovered" and P is *incomplete*.

If during a computation, P generates an infinite sequence of procedure calls, then P is *nonterminating*. Otherwise, it *terminates*.

We test for *partial correctness* and *completeness* by checking computation results against a program's specifications. *Termination* is tested for by routines that compare the inputs with respect to a specified well-founded ordering whenever a procedure is invoked.

5. AUTOMATED BUG LOCATION

When a Prolog program does not compute correct results, it may be that the program contains incorrect clauses, is incomplete in defining certain relationships between program variables, or has an infinite procedure invocation sequence. We turn now to discuss how each of these three types of errors can be detected and located automatically, based on the meta-programming capability of Prolog and executable specifications. When a program is submitted for debugging, we require that specifications of each procedure in the program be supplied, at least for those procedures that are likely to be wrong. In actual implementation, we can treat procedures without specifications as correct. Therefore, it is not necessary for the user to supply specifications if he is sure about the correctness of a procedure.

5.1. Locating Incorrect Clauses

If a program is partially correct then every subprogram of it is also partially correct, as the computation trees of a subprogram are a subset of those of the program as a whole. On the other hand, if a program terminates but returns an output which is incorrect with respect to its specifications, then this program contains at least one incorrect procedure.

5.1.1. The algorithm

Consider the computation of procedure $p(x',y')$ of program P with input x' and output y', with y' being incorrect with respect to the specifications of p. We trace the computation and check the result of each procedure call (by executing the specifications) as soon as it is completed. Suppose

$$q(u',v') :- r_1', \cdots, r_n'$$

is the first application (instance) of a clause to return an incorrect output v' on input u', then the clause being applied

$$q(u,v) :- r_1, \cdots, r_n$$

of procedure q is incorrect. This is explained by the fact that, if $q(u',v')$ is the first call returning an incorrect output, all the procedure calls $r_1', \cdots, r_n'$ must have completed earlier and returned correct results. Thus, the implication

$$q(u,v) :- r_1, \cdots, r_n$$

is false (for the instance u', v') with respect to the specifications.

The algorithm can be summarized as the following pseudo-Prolog code:

An Algorithm for Locating An Incorrect Clause

```
execute ( Goal , Message ) :-
        clause( Goal , Subgoal ),
        execute ( Subgoal , Message 1 ),
        diagnose ( Goal , Subgoal , Message , Message 1 )

diagnose ( Goal , Subgoal , ok ( Goal ), ok ( Subgoal ) ) :-
        spec ( Goal )
diagnose ( Goal , Subgoal , incorrect ( Goal :-Subgoal ), ok ( Subgoal ) ) :-
        not spec ( Goal )
diagnose ( Goal , Subgoal , Message 1, Message 1 )
```

The clause of *execute* is used to compute a goal (i.e., *Goal* in the algorithm above). It first finds a clause whose head can be unified with *Goal*, recursively solves the subgoals in the clause, and checks the results of computation. If all the subgoals return correct results, then we check if *Goal* is satisfied, by running specifications on the instantiated *Goal*. If the result is consistent with the specifications of *Goal*, then the clause is correct. The first clause of *diagnose* is for this purpose. If the computed *Goal* is not consistent with its specification, the second clause of *diagnose* will return an instance of the incorrect clause. If an error is identified in the subgoals, then the error message will be returned to the top level, using the third clause of *diagnose*.

5.1.2. An example

Consider the following insertion sort program, adapted (along with the other examples used in this section) from [Shapiro-83], with specifications for each of its procedures:

An Incorrect Insertion Sort		
isort ([*X* | *Xs*],*Ys*)	:-	*isort* (*Xs* ,*Zs*), *insert* (*X* ,*Zs* ,*Ys*)
isort ([ ],[ ])		
insert (*X* ,[*Y* | *Ys*],[*Y* | *Zs*])	:-	*Y* > *X* , *insert* (*X* ,*Ys* ,*Zs*)
insert (*X* ,[*Y* | *Ys*],[*X* ,*Y* | *Ys*])	:-	*X* ≤ *Y*
insert (*X* ,[ ],[*X*])		

Specifications for the Insertion Sort Program		
spec (*isort* (*X* ,*Y*))	:-	*ordered* (*Y*), *perm* (*X* ,*Y*)
spec (*insert* (*X* ,*Y* ,*Z*))	:-	*ordered* (*Y*) → *ordered* (*Z*), *perm* ([*X* | *Y*],*Z*)

The specification for *isort* is the same as that for *sort* in section 4 for generating test cases (actually this definition can be used for any sorting routine). For *insert*, the specification means that *insert* (*X* ,*Y* ,*Z*) holds if *Z* is in order and is a permutation of the list consisting of the element *X* and list *Y*, provided that *Y* is in order in the first place.

We now run *isort* on input [1,0,2] (the user of *Constructive Interpreter* actually need not supply the input list [1,0,2], since it can be generated by running the specifications of *isort*, as shown in previous section). Here is the result:

```
| ?- execute(isort([1,0,2],Answer), Message).

Error detected.   Debugging ...

The clause
    insert(0,[2],[2,0]):-gt(2,0),insert(0,[],[0])
is false!

Answer = X
Message = incorrect(insert(0,[2],[2,0]) :-
                        (gt(2,0),insert(0,[],[0])))

yes
```

We found a false instance of the first clause of *insert*. The error was due to an incorrect arithmetic test. Since the positions of the two arguments are exchanged, it forces a smaller element to be inserted after a larger element. The result is an unsorted list that fails on the specification check. (The variable *Message* is actually passed, in our debugging system, to the bug fixing routine which will be discussed in next section.)

To give a clearer picture of how this diagnostic procedure works, we use the following computation tree of *isort* on [1,0,2].

isort([1,0,2],[1,2,0])
isort([0,2],[2,0])
isort([2],[2]) *
isort([ ],[ ]) *
insert(2,[ ],[2]) *
insert(0,[2],[2,0]) ⇐
2 > 0 *
insert(0,[ ],[0]) *
insert(1,[2,0],[1,2,0])
1 ≤ 2

As we mentioned earlier, the diagnostic procedure traverses the computation tree in post-order and checks each procedure of its correctness. With reference to the above tree, during the diagnostic process each of the nodes marked with an asterisk has been verified by the interpreter as correct with respect to its specifications, while the node pointed by "⇐" is the first node that contains results inconsistent with its specifications. Therefore, the interpreter returns this node along with its two sons (equivalent to an instantiated clause) as a counterexample.

5.2. Locating Incomplete Procedures

A program P is said to be complete if for every procedure p in P, every possible input/output pair (x, y) implied by the specifications of p can be obtained from the execution of p. On the other hand, P is incomplete if there is a procedure p in P such that the execution of p, with input x, terminates, but fails to return the specified output y. This reflects the fact that the procedure defined for p is insufficient for computing the relations defined in the specifications of p. This incompleteness has been termed "finite failure" (cf. [Lloyd-84]) and corresponds to a computation tree which is finite but contains a node which represents an unsuccessful branch.

5.2.1. The algorithm

If P (finitely) fails on a procedure call $p(x', y)$ with legal input x' and uninstantiated output y (i.e., the specification of $p(x', y)$ is satisfiable), then P must contain at least one incomplete procedure. There are two possibilities: if p with input x' invokes no other procedures, then p is incomplete; if, on the other hand, p calls other procedures, then p or

one of the procedures invoked after p must be incomplete. Accordingly, we trace the execution of p. If a satisfiable call to a procedure q fails, while all procedures called by q return an answer whenever the call is satisfiable, then it is q that is deemed incomplete.

We summarize the above algorithm as follows:

An Algorithm for Locating Incomplete Procedures

```
execute ( Goal , Message ) :-
        clause( Goal , Subgoals ),
        spec ( Subgoals ),
        execute ( Subgoals , Message )

execute ( Goal , uncovered ( Goal ) ) :-
        spec ( Goals )
```

In other words, the interpreter for locating an incomplete procedure can be built in a way that it first tries to establish a computation tree from the execution of the goal and recursively executes the new subgoals. When a satisfiable call *Goal* fails to find a clause that can complete the computation, one can be sure that *Goal* is not covered, using the second clause of *execute*.

5.2.2. An example

Suppose we have an incomplete program as follows:

An Incomplete Insertion Sort

```
isort([X|Xs],Ys)          :-  isort(Xs,Zs),
                              insert(X,Zs,Ys)
isort([],[])

insert(X,[Y|Ys],[Y|Zs])   :-  X > Y,
                              insert(X,Ys,Zs)
insert(X,[Y|Ys],[X,Y|Ys])  :-  X ≤ Y
```

Given the same specifications as the ones for the incorrect insertion sort program, we try *isort* on [2,1,0]:

```
| ?- execute(isort([2,1,0],Answer), Message).
```

```
Error detected.   Debugging ...

The goal
    insert(0,[],[0])
is not covered!

Answer = X
Message = uncovered(insert(0,[],[0]))

yes
```

We now have an instance of the uncovered goal and the debugger detects that the incomplete procedure is *insert*, which does not have a clause to cover the base case (when inserting an element to an empty list).

The incomplete computation tree of *isort* on [2,1,0] is:

$$
\begin{array}{l}
isort([2,1,0],Answer) \\
\quad isort([1,0],X_1) \\
\quad\quad isort([0],X_2) \\
\quad\quad\quad isort([\,],[\,]) \\
\quad\quad insert(0,[\,],X_2) \quad \Leftarrow
\end{array}
$$

In the pre-order traversal of the tree, $insert(0,[\,],X_2)$ is the first goal that cannot be unified with any clause in the program. The computation stops at this point because of the failure of this node.

5.3. Locating Diverging Procedures

5.3.1. The algorithm

If ***P*** is partially correct, but nonterminating, then during the computation, some procedure *p* must be invoked repeatedly (though, there may be calls to other procedures in between the calls to *p*), with the sequence of input values to *p* not decreasing in the specified well-founded ordering > for *p*. In the computation tree, a diverging computation corresponds to infinite growth on one branch. This nonterminating computation can be detected by tracing ***P*** and checking that each call is smaller with respect to > than the previous one.

This algorithm can be summarized as:

An Algorithm for Locating Diverging Procedures

```
execute ( Goal , Message ) :-
        well–founded ( Goal )
        clause( Goal , Subgoal ),
        execute ( Subgoal , Message 1 ),
        diagnose ( Goal , Subgoal , Message , Message 1 )

execute ( Goal , looping ( Goal ) )
```

That is, before executing a goal, the interpreter first checks if the values of input variables violate the well-founded ordering defined for that goal. This can be done by examing previous procedure calls in the run time stack and compare values of input variables for consecutive calls to the same procedure. If the input values do get smaller for later calls, then the call will not result in looping and, therefore, the interpreter can just go ahead and execute the goal as usual. On the other hand, if the well-founded ordering is violated, the interpreter will report an error message.

5.3.2. An example

Given the following example with its well-founded ordering specifications:

A Looping Insertion Sort

```
isort([X|Xs],Ys)          :-  isort(Xs,Zs),
                              insert(X,Zs,Ys)
isort([ ],[ ])

insert(X,[Y|Ys],[Y|Zs])   :-  insert(X,Ys,Ws),
                              insert(Y,Ws,Zs)
insert(X,[Y|Ys],[X,Y|Ys]) :-  X ≤ Y
insert(X,[ ],[X])
```

Well-founded Ordering for Recursive Procedures

```
wfo(isort(X,Y),isort(U,V))   :-  shorter(X,U)
wfo(insert(X,Y,Z),insert(U,V,W))  :-  shorter(Y,V)
```

The predicate *wfo* specifies the well-founded ordering for sequences of input values. For both *isort* and *insert*, the number of elements in the input list should decrease with each recursive call. As with the case for predicates *perm* and *ordered*, *shorter* can be defined in

Prolog in a straightforward, inefficient way such as:

```
shorter(X,Y)   :-   length(X,Nx),
                    length(Y,Ny),
                    Nx < Ny
```

Running *isort* on [1,0,2], we have

```
| ?- execute(isort([1,0,2],Answer), Message).

Error detected.   Debugging ...

The goal  "insert(2,[0],X)"  in the clause
    insert(0,[2],X):-insert(0,[],[0]),insert(2,[0],X)
is looping!

Answer = Y
Message = looping(insert(0,[2],X))

yes
```

As can be seen, the second argument of the goal *insert*(2,[0],*X*) has the same length as the second argument of the head of the clause, and thus violates the relationship defined in *wfo* (*insert*).

5.4. A Meta-Interpreter for Automatic Bug Location

We can combine the features of the different aspects of debugging illustrated in this and the previous section into one system. This diagnostic interpreter can be summarized as:

An Automatic Meta-Interpreter for Bug Location

```
execute ( (Goal 1, Goal 2), Message ) :-
    execute ( Goal 1, Msg_Goal 1 ),
    if Msg_Goal 1 = ok( Goal 1 )
        then execute ( Goal 2, Message )
        else Message = Msg_Goal 1

execute ( Goal, ok(Goal) ) :-
    system( Goal ), Goal

execute ( Goal, looping(Goal) ) :-
    not_decreasing ( Goal )

execute ( Goal, Message ) :-
    clause( Goal, Subgoals ),
    execute ( Subgoals, Msg_Subgoal ),
    if Msg_Subgoal = ok( Subgoals )
        then if spec ( Goal )
                then Message = ok( Goal )
                else Message = incorrect( (Goal :- Subgoals) )
        else Message = Msg_Subgoal

execute ( Goal, uncovered(Goal) )
```

The procedure *execute* (*Goal*, *Message*) serves two functions: goal reduction and bug location. The first clause deals with conjunctive goals. It first executes the first conjunct. If the result is correct, it then goes on executing the remaining conjuncts; otherwise, it just returns the error found to the top level. The second clause is for built-in primitives: they are executed directly by the system. The next three clauses of the procedure detect bugs of nontermination, incorrect clauses, and uncovered goals, respectively. The predicate *not_decreasing* checks if the input variables of *Goal* violate the well-founded ordering defined in the specification of the procedure that covers the goal, if such is the case, we have an instance of a looping goal. If the input cannot cause an infinite sequence of procedure calls, the interpreter will proceed to check if the program can actually complete the computation on the given input. It first finds a clause whose head can be unified with *Goal* and then recursively executes (and debugs) the subgoals in the body of that clause. If a bug is found in the body of a clause, it will be returned to the top level for correction. If all the subgoals complete successfully, then all the output variables in *Goal* will be instantiated. The interpreter then checks if the output value is correct with respect to the specifications of *Goal*. If not, then we have found an incorrect clause. On the other hand, if there is no clause in the program that covers the goal for the input data (i.e., no unifying clause or a subgoal fails in every unifying clause), then, since *Goal* is satisfiable according to the specifications, the program must be incomplete and we have an instance of an uncovered goal.

6. HEURISTIC BUG CORRECTION

Just as knowing that a program is incorrect does not mean that one knows where the bug is, knowing the location of a bug does not imply that one knows how to correct it. Although Myers [Myers-79] has claimed that 95% of the debugging effort is spent on bug location and that bug correction is a much easier task than bug location, we believe that correcting a bug after it is identified is generally a more difficult task than locating the bug, especially when it is to be performed by a machine. This is because bug location only requires tracing the execution of procedures and checking the results of computation. Bug correction, on the other hand, requires reasoning with knowledge of the domain and intended algorithm, the semantics of the programming language and the input/output specifications.

In the automation process, it is an intricate task to formalize the complex knowledge involved in bug correction and represent it in a form that can be utilized by the debugger. Some automatic debugging systems (e.g., [Murray-86]) use the stored information in their system's knowledge base for bug correction by matching (maybe partially) and replacing the buggy program with the established code fragments or algorithms. In our case, we have only the knowledge contained in the specifications of the individual procedures and the operational semantics of pure Prolog. In addition, we have devised some heuristics — based on a classification of Prolog bugs — that suggest a possible cause for the error. Deductive or inductive corrective measures (or both) are then employed in an attempt to bring the program in line with the given specifications.

6.1. Fixing an Incorrect Clause

A clause

$$p(x,y) :- p_1, \cdots, p_n$$

is incorrect if there is an instance of that clause, say,

$$p(x',y') :- p_1', \cdots, p_n'$$

such that all the p_i''s are true (i.e., their specifications hold), but $p(x',y')$ is false. (Here x' denotes the test input value(s) to p and y' is the output after the call $p(x',y)$ returns.) To fix this incorrect clause, we first rerun the specification of p to get a correct output, say y'', for the given input x'. How the program behaves with the goal $p(x',y'')$ will help guide the debugger.

If the *solved* goal $p(x',y'')$ is covered by another clause in the program (i.e., there exists at least one clause in the procedure that computes this goal correctly), then the incorrect clause should not have completed and returned a wrong result. Instead, the clause should presumably have failed for this input. We can, therefore, attempt to include extra conditions that prevent computation for the improper input x'. To add subgoals to the clause, we try to construct a proof that the right hand side of the clause implies the left hand side. If the proof fails because of some missing conditions, we can add them as subgoals to the clause (detail below). Alternatively, we can use the offending clause as a starting point for an inductive synthesis of a correct clause (see below). In the worst case, we can always add the subgoal **fail** to the clause. Although this might be too strong a fix and might result in some other goals becoming uncovered, adding **fail** as a subgoal does make the clause

(vacuously) correct. We will discuss below how to deal with any uncovered goals.

If the solved goal $p(x',y'')$ is only covered by the incorrect clause, then we proceed to add conditions that preclude computation of the wrong answer y', with input x', as above. A sufficient condition (viz. if $x = x'$ then $y = y''$) can be deduced from the variable bindings obtained when unifying $p(x',y'')$ with the clause head $p(x,y)$ and may be added to the clause as subgoals. Or, an inductive approach may be taken.

If the *solved* goal $p(x',y'')$ is not covered by any clause, then the fix proceeds in different directions, depending on whether $p(x',y'')$ can be unified with the head of the incorrect clause. If the head does unify, but some of the subgoals fail for y'', then we presume that the incorrect clause should cover the goal $p(x',y)$ and compute y'' instead of y'. In this case, we can combine fixes for the uncovered goal, $p(x',y'')$, and the incorrect clause that computes the erroneous solution $p(x',y')$. We check, for $p(x',y'')$ (i.e., under the current input and *correct* output), which of the subgoals in the clause fail with the output constrained to be y''. After identifying any such incorrect subgoals, we try to fix them by either applying a heuristic rule or an inductive method. We rearrange, replace, delete, or add new variables within subgoals until the original incorrect clause computes $p(x',y'')$ correctly. The induction method that we use to correct incorrect subgoals is a modification of the refinement method in Shapiro's Model Inference System [Shapiro-83]. (A detailed analysis of these methods follows in a later section.)

The last possibility is that $p(x',y'')$ cannot be unified with the head of the incorrect clause, nor is it covered by other clauses in the program. In this case, we assume that the incorrect clause we have identified should cover this goal. Accordingly, the only way to correct the bug is to first fix (i.e., weaken) the clause head so that it is unifiable with $p(x',y'')$. The methods described above can then be used to fix any incorrect subgoals.

We summarize the strategies for correcting an incorrect clause as the following heuristic rules:

(1) **If** the solved goal is covered by a clause in the program, **then** deduce missing subgoals and add them to the incorrect clause to preclude the wrong answer.

(2) **If** the solved goal can be unified with the head of the incorrect clause **but** is not covered by any clause in the program, **then** fix the subgoals that fail for the correct answer and continue debugging the clause.

(3) **If** the solved goal cannot be unified with the head of the incorrect clause **and** is not covered by any clause in the program, **then** fix the clause head and continue debugging the clause.

6.1.1. An example

Given the following insertion sort program:

An Incorrect Insertion Sort		
isort ([*X* |*Xs*],*Ys*)	:-	*isort* (*Xs* ,*Zs*), *insert* (*X* ,*Zs* ,*Ys*)
isort ([ ],[ ])		
insert (*X* ,[*Y* |*Ys*],[*Y* |*Zs*])	:-	*insert* (*X* ,*Ys* ,*Zs*)
insert (*X* ,[*Y* |*Ys*],[*X* ,*Y* |*Ys*])	:-	$X \leq Y$
insert (*X* ,[ ],[*X*])		

We now test the program on input list [0,1]:

```
| ?- debug(isort([0,1],Answer)).

Error detected.   Debugging ...

The clause
    insert(0,[1],[1,0]):-insert(0,[],[0])
is false!

The goal "insert(0,[1],[0,1])" is covered
There are missing subgoals in the clause:
     insert(X,[Y|Z],[Y|V]):-insert(X,Z,V)
Retract erroneous clause:
     insert(X,[Y|Z],[Y|V]):-insert(X,Z,V)

Generating missing subgoals ...
Assert clause:
    insert(X,[Y|Z],[Y|V]):-Y<X,insert(X,Z,V)
```

The debugger detected an incorrect clause in procedure *insert* when trying to solve the goal *isort* ([0,1],*Answer*). After some analysis, it determined that the clause

insert (*X* ,[*Y* |*Z*],[*Y* |*V*]) :- *insert* (*X* ,*Z* ,*V*)

is false for $X=0$, $Y=1$, $Z=[\,]$, $V=[0]$ (note that the debugger occasionally renames variables); furthermore, it need not be covering the subgoal *insert* (0,[1],*Z*), since the solved subgoal *insert* (0,[1],[0,1]) is in fact covered by another clause,

insert (*X* ,[*Y* |*Z*],[*X* ,*Y* |*Z*]) :- $X \leq Y$,

in the program. The debugger then tried to deduce a missing subgoal by constructing a proof. It tried to prove that *insert* (*X* ,*Z* ,*V*) implies *insert* (*X* ,[*Y* |*Z*],[*Y* |*V*]), and concluded that, by adding $Y < X$ to the right-hand side of the clause, the implication holds. Therefore, the debugger removed the incorrect clause and asserted the synthesized clause to the program. We will detail this proof process in a later section.

6.2. Fixing an Incomplete Program

To remedy the problem of an uncovered goal, we first check if the goal can be unified with the head of a clause. If indeed such a clause exists, then we presume that it should cover this goal. Since the original clause might be useful for other goals, instead of modifying the clause directly, we make local changes on a copy. We locate the subgoal that causes this clause to fail and either try to fix it inductively (by rearranging, replacing, deleting, or adding variable within the subgoal) or eliminate the offending subgoal entirely and use deductive means to correct it, if necessary.

When there is no clause whose head unifies with the uncovered goal, we use the specifications to synthesize a new clause. This can be done by using the uninstantiated goal as the clause head and the specifications as the clause body, simplifying the resulting clause as much as possible, or by an inductive method, using the specifications to guide the search. We can also fix a clause head so that it can be unified with the uncovered goal, and then debug the subgoals in the clause.

The above strategies for dealing with uncovered goals can be summarized as follows:

(1) **If** the uncovered goal can be unified with the head of a clause, **then** duplicate the clause, and locate and fix its unsatisfiable subgoals.

(2) **If** the uncovered goal cannot be unified with the head of a clause, **then** use the specifications for that goal to synthesize a new clause.

6.3. Fixing a Looping Procedure

When the input to a procedure call violates the well-founded ordering defined for that procedure, a likely cause is that the input argument of the call is too general. For example, it may contain an irrelevant variable that does not appear in either the clause head or other subgoals of the same clause. Other possibilities are that some variables are missing or that the order of arguments is wrong. In any of these cases, what we have is a clause that contains a looping call caused by incorrect arguments. We try to fix the offending subgoal, using the same inductive method as for fixing incorrect subgoals. Alternatively, we can weaken it and employ deductive techniques to ensure that the well-founded condition is met.

It is also possible that a subgoal that would preclude the looping case is missing (and that the goal is covered by another clause). This can be treated in the same way as an incorrect clause.

6.4. Deducing Missing Subgoals

According to the deductive semantics of Prolog, the right hand side (the body) of a clause should imply the left hand side (the head). That is, in proving the correctness of a correct clause, the implication should be found to hold. On the other hand, trying to prove

the implication for an incorrect clause must result in failure. Therefore, we can try to prove the head of the clause, given the subgoals in the body as hypotheses, and in the process identify and derive those sufficient conditions that will allow a proof to go through. (Unlike some methods such as that in [Katz-Manna-75], a correct clause would never be "debugged"; only a clause found faulty by testing is subjected to formal verification.)

This approach is inspired by the work of [Smith-82] in which a deductive theorem prover was used to derive a sufficient precondition such that a goal can be shown to logically follow from the conjunction of the precondition and a hypothesis. In other words, the precondition provides any additional conditions under which a goal can be proved from a hypothesis. We adopted and modified this method and constructed a theorem prover for Horn clauses.

6.4.1. A theorem prover for Horn clauses

The deductive proof proceeds by reducing both sides of the clause to be proved to simpler forms, by replacing each goal (or subgoal) with its definitions or with something that implies it, and each hypothesis with its definition or something that it implies, until a primitive goal is obtained or the proof fails (details in a later paragraph).

The prover employs the following rules which, for the most part, are modifications of typical rules for deductive proof (cf., [Loveland-78]). In the rules we use G (possibly with a subscript or superscript) to represent a goal, H (possibly with a subscript) for a hypothesis, $\&$, $\vee$, and $\sim$ for logical "and", "or", and "not", respectively, "$H \rightarrow G$" for "if H then G", and "*lhs* $\Rightarrow$ *rhs*" for "given *lhs* (left hand side), it is sufficient to prove *rhs* (right hand side)."

(1) Eliminate the hypothesis
Rule 1: $H \rightarrow$ **true** $\Rightarrow$ **true**.
Explanation: If the goal can be reduced to **true**, the implication will hold and the proof can be regarded as successful.

(2) Eliminate the goal
Rule 2: false $\rightarrow G \Rightarrow$ **true**.
Explanation: If the hypothesis can be reduced to **false**, the implication is vacuously correct.

(3) Eliminate the goal which is a subset of the hypothesis
Rule 3: $H \rightarrow G \Rightarrow$ **true, if** $G \subset H$.
Explanation: If the goal is reduced to a subset of the hypothesis, then the implication is established.

(4) Eliminate the hypothesis which is a subset of the goal
Rule 4: $H \rightarrow G \Rightarrow G'$, **if** $G' = G - H$.
Explanation: If the hypothesis is a subset of the goal, it is sufficient to prove the subgoals that are not hypothesis.

(5) Reduce a conjunctive goal
Rule 5: $H \rightarrow G_1 \,\&\, G_2 \quad \Rightarrow \quad (H \rightarrow G_1) \,\&\, (H \rightarrow G_2)$.
Explanation: To prove a conjunctive goal, one needs to prove each conjunct separately.

(6) Reduce a disjunctive goal
Rule 6: $H \rightarrow G_1 \vee G_2 \quad \Rightarrow \quad (H \rightarrow G_1) \vee (H \rightarrow G_2)$.
Explanation: To prove a disjunctive goal, one needs to prove only one of the disjuncts.

(7) Reduce a disjunctive hypothesis
Rule 7: $(H_1 \vee H_2) \rightarrow G \quad \Rightarrow \quad (H_1 \rightarrow G) \,\&\, (H_2 \rightarrow G)$.
Explanation: To prove a goal with disjunctive hypotheses, one can prove that the goal can be proved from each disjunct.

(8) Reduce an implicative goal
Rule 8: $H \rightarrow (G_1 \rightarrow G_2) \quad \Rightarrow \quad (H \,\&\, G_1) \rightarrow G_2$.
Explanation: To prove a goal which is an implication itself, include the precondition of the implication as part of the hypothesis and prove the postcondition of the implication.

(9) Reduce an implicative hypothesis
Rule 9: $(H_1 \rightarrow H_2) \rightarrow G \quad \Rightarrow \quad (\sim H_1 \rightarrow G) \,\&\, (H_2 \rightarrow G)$.
Explanation: To prove a goal with an implicative hypothesis, prove the goal with the negation of the precondition of the implication, and with the postcondition of the implication, respectively.

(10) Take contrapositive
Rule 10: $\sim H \rightarrow \sim G \quad \Rightarrow \quad G \rightarrow H$.
Explanation: If both the hypothesis and the goal are in negation form, then one can drop both negations and reverse the hypothesis and the goal for the proof.

(11) Take contrapositive – generalized rule
Rule 11: $\sim H_1 \,\&\, H_2 \rightarrow \sim G \quad \Rightarrow \quad G \,\&\, H_2 \rightarrow H_1$.
Explanation: If the goal and one part of the hypothesis are in negation form, then the proof can be established if one can show that the negation part of the hypothesis can be derived from the negation of the goal and the non-negation part of the hypothesis combined.

In addition to these proof rules, there are three ways of reducing a goal or subgoal. First, we can replace the goal with its definition as described in the goal's specification. This is substitution of equivalent terms:

$$H \rightarrow G \quad \Rightarrow \quad H \rightarrow G', \text{ if } G = G'.$$

It is obvious that, if one substitutes the goal with equivalent terms, the proof condition will remain the same. Second, if there is a correct program clause whose head matches the goal, we can replace the goal with the subgoals in that clause. This is just like goal reduction in normal Prolog computation. It can also be regarded as the application of implicative terms:

$$H \rightarrow G \quad \Rightarrow \quad H \rightarrow G', \text{ if } G \rightarrow G'.$$

Third, if a specific domain fact is known, it can be used to weaken a goal or replace it with something equivalent (e.g., replacing a list with one of its permutations when the order does

not affect the truth value of the predicate). This is an effort to build into the debugger a knowledge handling capability such that it can have some general knowledge when reasoning about programs.

Similarly, we can also apply the following rules for hypothesis reduction. First, substituting the hypothesis with equivalent terms:

$$H \rightarrow G \quad \Rightarrow \quad H' \rightarrow G, \text{ if } H = H'.$$

If we substitute the hypothesis with equivalent terms, the proof condition remains the same. Second, substituting the hypothesis with something that implies the hypothesis:

$$H \rightarrow G \quad \Rightarrow \quad H' \rightarrow G, \text{ if } H' \rightarrow H.$$

It is obvious that this is true.

The proof of a clause starts with the head of the clause as initial goal and the body of the clause as initial hypothesis. It keeps applying the proof rules until one of the following conditions is met: (1) the initial goal is reduced to **true**, in which case the clause is proved correct; (2) the initial hypothesis is reduced to **false**, meaning that there are conflicting subgoals in the clause, and that the clause is vacuously correct; (3) the goal is reduced to a subset of the hypotheses, in which case the implication is also established; and (4) the original goal is reduced to primitives and hypotheses, in which case those goals not appearing as hypotheses are added as subgoals to the original clause. If the proof ends in condition (4), then we have identified those missing subgoals that will make the clause correct.

The goal reduction mechanism is equipped with a logical simplifier (cf., [Waldinger-Levitt-74]). This simplifier is invoked after each reduction step and performs tasks such as removing nested conjunctions, duplicate goals, and tautologies (i.e., the goal **true**). It also simplifies the goal structures according to the logical rules governing **and**, **or**, **not**, and **implication**. For example, if a conjunctive goal contains the subgoal **false**, then the whole goal will be reduced to **false**. Also, for the purpose of hypothesis reduction, we assume that the Horn clauses possess the power of equivalent definitions (cf. [Kowalski-79]).

6.4.2. An example

We now present an example of how a deductive proof can be used to derive missing subgoals which will make the clause correct. Suppose we have identified the following clause as being incorrect:

$$insert(X,[V|Y],[V|Z]) \text{ :- } insert(X,Y,Z).$$

We try to prove the head $insert(X,[V|Y],[V|Z])$, using the body $insert(X,Y,Z)$ as a hypothesis. In implementing this prover, we skolemize all the variables in the clause, since otherwise Prolog will unify variables which is undesirable when constructing a proof. The proof condition is thus:

(1)	Current Hypothesis	Current Goal
	insert (*x* ,*y* ,*z*)	*insert* (*x* ,[*v* |*y*],[*v* |*z*])

We first replace the hypothesis with the specification for *insert* (*x* ,*y* ,*z*) (given in section **5.1.2.**) and obtain

(2)	Current Hypothesis	Current Goal
	ordered (*y*) → *perm* ([*x* |*y*],*z*), *ordered* (*z*)	*insert* (*x* ,[*v* |*y*],[*v* |*z*])

Using the rule of reduction of implicative hypothesis, the proof can be split into two parts:

(3)	Current Hypothesis	Current Goal
	perm ([*x* |*y*],*z*) *ordered* (*z*)	*insert* (*x* ,[*v* |*y*],[*v* |*z*])

and

(4)	Current Hypothesis	Current Goal
	~*ordered* (*y*)	*insert* (*x* ,[*v* |*y*],[*v* |*z*])

We first try to prove condition (3). Since no other rules apply at this point, we replace the goal with its specification:

(5)

Current Hypothesis	Current Goal
perm ([*x* | *y*],*z*)	*ordered* ([*v* | *y*]) →
ordered (*z*)	*perm* ([*x* ,*v* | *y*],[*v* | *z*]),
	ordered ([*v* | *z*]))

Applying the rule of reduction of implicative goal, the precondition of the goal becomes part of the hypothesis; therefore, we need to prove

(6)

Current Hypothesis	Current Goal
ordered ([*v* | *y*])	*perm* ([*x* ,*v* | *y*],[*v* | *z*])
perm ([*x* | *y*],*z*)	*ordered* ([*v* | *z*])
ordered (*z*)	

Now that we have a conjunctive goal, the proof is divided into two parts. The proof condition for the first conjunct is

(7)

Current Hypothesis	Current Goal
ordered ([*v* | *y*])	*perm* ([*x* ,*v* | *y*],[*v* | *z*])
perm ([*x* | *y*],*z*)	
ordered (*z*)	

At this point, no reduction rule can be applied to the proof. However, there is a clause in the procedure of *ordered*, namely,

ordered ([*H* \| *T*]) :- *lt_all* (*H* ,*T*), *ordered* (*T*),

whose head can be unified with *ordered* ([*v* \| *y*]) (literally, the above clause means that the list [*H* \| *T*] is in order if the head *H* is smaller than every element of the tail *T*, and *T* is in order). This part of the hypothesis can, therefore, be reduced to the two subgoals in the body of that clause. After unifying *H* with *v* and *T* with *y* we have

(8)	Current Hypothesis	Current Goal
	lt_all (*v* ,*y*) *ordered* (*y*) *perm* ([*x* |*y*],*z*) *ordered* (*z*)	*perm* ([*x* ,*v* |*y*],[*v* |*z*])

We next utilize one of the domain facts: if list L_1 is a permutation of list L_2, then after removing a common element from both lists, the remaining lists are still permutations of each other. Since, in the current goal, list [*x* ,*v* |*y*] is a permutation of [*v* |*z*] and both contain the element *v* , by removing *v* from both lists, [*x* |*y*] is still a permutation of *z* . The goal can thus be replaced by something simpler and the proof condition becomes

(9)	Current Hypothesis	Current Goal
	lt_all (*v* ,*y*) *ordered* (*y*) *perm* ([*x* |*y*],*z*) *ordered* (*z*)	*perm* ([*x* |*y*],*z*)

We see that the goal is now part of the hypothesis, which is one of the terminating conditions for the proof. Therefore, we have shown that the first conjunct of the goal in step (6) can be obtained, without the necessity of adding any preconditions. For the second conjunct, the proof condition is

(10)	Current Hypothesis	Current Goal
	ordered ([*v* |*y*]) *perm* ([*x* |*y*],*z*) *ordered* (*z*)	*ordered* ([*v* |*z*])

We again use a clause for *ordered* to reduce the hypothesis, as in step (7):

(11)

Current Hypothesis	Current Goal
lt_all (*v* ,*y*)	*ordered* ([*v* |*z*])
ordered (*y*)	
perm ([*x* |*y*],*z*)	
ordered (*z*)	

The same clause can also be used to weakened the goal, since it has the same predicate and variable structure. We therefore have

(12)

Current Hypothesis	Current Goal
lt_all (*v* ,*y*)	*lt_all* (*v* ,*z*)
ordered (*y*)	*ordered* (*z*)
perm ([*x* |*y*],*z*)	
ordered (*z*)	

This conjunctive goal needs to be processed in two parts. First,

(13)

Current Hypothesis	Current Goal
lt_all (*v* ,*y*)	*lt_all* (*v* ,*z*)
ordered (*y*)	
perm ([*x* |*y*],*z*)	
ordered (*z*)	

Since, from the hypothesis, we know that z is a permutation of the list $[x|y]$, and the goal *lt_all* (v,z) only requires that v is smaller (less) than all the elements in z, regardless of the order, we can substitute z with $[x|y]$ without changing the truth value of *lt_all* (v,z). This substitution gives us the following condition:

(14)	Current Hypothesis	Current Goal
	lt_all (*v* ,*y*)	*lt_all* (*v* ,[*x* |*y*])
	ordered (*y*)	
	perm ([*x* |*y*],*z*)	
	ordered (*z*)	

We now have another program clause that we can use to reduce current goal:

lt_all (*E* ,[*H* \|*T*]) :- *E* < *H* , *lt_all* (*E* ,*T*)

(this clause means that E is smaller than all the elements in list $[H|T]$ if E is smaller than the head H and is smaller than every element in tail T). Therefore (unifying E with v, H with x, and T with y), we have yet another conjunctive goal to prove:

(15)	Current Hypothesis	Current Goal
	lt_all (*v* ,*y*)	*v* < *x*
	ordered (*y*)	*lt_all* (*v* ,*y*)
	perm ([*x* |*y*],*z*)	
	ordered (*z*)	

The first conjunct happens to contain a primitive "<." Since this goal cannot be derived from the hypothesis and meets the condition for terminating the proof, we record it as a condition that needs to be added to the original hypothesis, if we want the proof to be successful. The second conjunct actually appears in the hypothesis and is, therefore, proven. In consequence, the condition in step (15) is satisfied, with the realization that "$v < x$" should be added as a subgoal in the original clause.

In step (12), the second conjunct is satisfied, since it is part of the hypothesis:

(16)	Current Hypothesis	Current Goal
	lt_all (*v* ,*y*)	*ordered* (*z*)
	ordered (*y*)	
	perm ([*x* |*y*],*z*)	
	ordered (*z*)	

Back to the second part of the proof from step (2), we proceed from step (4) and

replace the goal with its specification, resulting in the following condition:

(17)

Current Hypothesis	Current Goal
$\sim ordered(y)$	$ordered([v \mid y]) \rightarrow$ $perm([x, v \mid y], [v \mid z]),$ $ordered([v \mid z])$

Applying the rule of reduction of implicative goal, the precondition of the goal becomes part of the hypothesis:

(18)

Current Hypothesis	Current Goal
$ordered([v \mid y])$ $\sim ordered(y)$	$perm([x, v \mid y], [v \mid z])$ $ordered([v \mid z])$

With a conjunctive goal, we try to prove the first conjunct first. After reducing $ordered([v \mid y])$ (same as in step (7)), we have

(19)

Current Hypothesis	Current Goal
$lt_all(v, y)$ $ordered(y)$ $\sim ordered(y)$	$perm([x, v \mid y], [v \mid z])$

Since there are complementary terms ($ordered(y)$ and $\sim ordered(y)$) in the hypothesis, after simplifying, the proof condition is now

(20)

Current Hypothesis	Current Goal
false	$perm([x, v \mid y], [v \mid z])$

This is, of course, trivially satisfied. The second conjunct of the goal in step (18) can also be satisfied in the same way.

In summary, during the whole deductive proof process, we have shown that the original incorrect clause satisfies its property of implication, except in step (15) where we identify a missing condition (i.e., $v < x$) that must be added as a subgoal (hypothesis). If we include this subgoal in the original clause, we will have a successful proof.

6.5. Fixing Incorrect Subgoals

Once we identify an incorrect subgoal, we can correct it using either a heuristic rule or an inductive method, besides using the deductive methods outlined in the previous sections.

We have developed heuristics that are meant to correct an incorrect subgoal quickly when a certain pattern of subgoals is encountered. For example, one of the rules is to swap the variables if there are only two variables in the subgoal. Other rules include moving a simple variable to a different position, replacing simple variables with more complicated terms, deleting seemingly redundant variables, and adding free variables that have appeared elsewhere in the same clause. The purpose of this kind of heuristic is to attempt to fix some commonly made, yet easily corrected, errors. These heuristics were lifted from Shapiro's work.

When our heuristic rules cannot correct the errors in a subgoal, a general inductive strategy will be employed with the hope of fixing the bugs. This is done by applying refinement operations on terms within the subgoals. For example, we can try to unify two free variables, or unify a compound term with variables appearing elsewhere in the same clause.

It should be noted that all heuristic fixes will be tested immediately after the changes are made; and if the fixes do not correct the errors, all the changes are undone.

7. THE CONSTRUCTIVE INTERPRETER

The functions of test case generation, bug discovery, bug location, and bug correction can be integrated in one automated debugging environment. Our realization of this framework is named the *Constructive Interpreter*. The structure of this interpreter can be described in pseudo-Prolog code as follows:

The Constructive Interpreter
interpret (*Goal* (*Input* ,*Output*)) :- *spec* (*Goal* (*Input* ,*Output*)), *skolemize* (*Input* ,*Skolem*), *execute* (*Goal* (*Skolem* ,*Output*), *Message*), *fix_bug* (*Message*)

Upon receiving a goal, the interpreter first examines the input variables. If the input is symbolic, then by executing the specifications of the procedure, the interpreter will generate test cases. Running the specifications on the given or generated input also checks that the input values are satisfiable. Once the legality of the input is established or a legal test input generated, the interpreter proceeds to execute the program on skolemized input. (Skolemization forces the program to find one symbolic output for all inputs with the same given structure.) If execution completes successfully, the interpreter returns correct output values. In the case of symbolic input, the user can continue to generate alternate test cases and execute the program on different inputs. If ever the execution fails, i.e., if the program contains an incorrect, incomplete, or nonterminating procedure, then the interpreter will locate a bug and return a diagnostic message. Bug-fixing routines will then be invoked to correct the bug that has been identified and located.

The procedure *execute* does goal reduction and bug location, and has been discussed in section 5. The procedure *fix_bug* (*Message*) implements the bug correction heuristics discussed in section 6.

This interpreter is constructive in the sense that it assumes an active role during the debugging process and actually tries to complete the construction of the program being debugged, all with very little user involvement. As will be seen in the next section, this interpreter can also be used to synthesize Prolog programs.

7.1. Implementation

The *Constructive Interpreter* is written in C-Prolog and runs in a 4.3 BSD Unix environment. It is based on the meta-interpreter introduced in section 3.

The three major components are test case generator, bug locator, and bug corrector. The test case generator executes specifications to either generate test input or verify the satisfiability of user-supplied input. The bug locator also carries out the computation. It has a run-time stack that records all procedure invocations. This information and the specified well-founded ordering are used to check against looping. The execution is simulated to perform depth-first search and backtracking upon failure. A message stack is maintained during execution, and an error message is recorded whenever an error occurs.

The bug corrector contains three main procedures, dealing with three different kind

of errors respectively. In addition to performing error analysis and suggesting fixes, they all have access to the deductive theorem prover and inductive subgoal refiner.

7.2. Examples

In this section, we illustrate the integrated functions, including test case generation, bug location, and correction, of the *Constructive Interpreter*. Our experimental implementation is able to generate test cases that reveal errors and locate bugs for all the sorting examples in [Shapiro-83].

The following is an annotated script of the *Constructive Interpreter* debugging a quicksort program.

A Buggy Quick Sort Program		
qsort ([*X* |*L*],*L* 0)	:-	*part* (*L* ,*X* ,*L* 1,*L* 2), *qsort* (*L* 1,*L* 3), *qsort* (*L* 2,*L* 4), *append* ([*X* |*L* 3],*L* 4,*L* 0)
part ([*X* |*L*],*Y* ,*L* 1,[*X* |*L* 2])	:-	*part* (*L* ,*Y* ,*L* 1,*L* 2)
part ([*X* |*L*],*Y* ,[*X* |*L* 1],*L* 2)	:-	*X* ≤ *Y* , *part* (*L* ,*Y* ,*L* 1,*L* 2)
part ([ ],*X* ,[*X*],[ ])		
append ([*X* |*L* 1],*L* 2,[*X* |*L* 3])	:-	*append* (*L* 1,*L* 2,*L* 3)
append ([ ],*L* ,*L*)		

Specifications for the Quick Sort Program		
spec (*qsort* (*X* ,*Y*))	:-	*ordered* (*Y*), *perm* (*X* ,*Y*)
spec (*part* (*L* ,*E* ,*X* ,*Y*))	:-	*rm_list* (*X* ,*L* ,*Y*), *gt_all* (*E* ,*X*), *lt_all* (*E* ,*Y*)
spec (*append* (*X* ,*Y* ,*Z*))	:-	*length* (*X* ,*N*), *front* (*N* ,*Z* ,*X*), *rm_list* (*X* ,*Z* ,*Y*)
wfo (*qsort* (*X* ,*Y*), *qsort* (*U* ,*V*))	:-	*shorter* (*X* ,*U*)
wfo (*part* (*X* ,*A* ,*B* ,*C*), *part* (*Y* ,*D* ,*E* ,*F*))	:-	*shorter* (*X* ,*Y*)
wfo (*append* (*X* ,*A* ,*B*), *append* (*Y* ,*C* ,*D*))	:-	*shorter* (*X* ,*Y*)

The specifications say that *qsort* (*X* ,*Y*) holds if *Y* is sorted and *Y* is a permutation of *X* , that

part(*L*,*E*,*X*,*Y*) holds if *Y* is the list obtained by removing elements of *X* from *L* (in other words, *L* is a permutation of *X* and *Y* combined) and *E* is greater than all the elements in *X* and smaller then all the elements in *Y*, and that *append*(*X*,*Y*,*Z*) is true if *Z* is the combination of lists *X* and *Y*, in their original order. The predicate *wfo* specifies the well-founded ordering for sequences of input values. For all procedures *qsort*, *part*, and *append* the number of elements in the input list should decrease with each recursive call. As is also the case for the insertion sort program, the predicates *perm*, *ordered*, *rm_list*, *gt_all*, *lt_all*, and *shorter* can be defined as usual Prolog procedures. (These procedures should be regarded as standard building blocks for specification, available in the debugger's library, since they all apply across a whole gamut of specific programs. For example, *lt_all* would play a role in virtually all sorting and most searching programs and *rm_list* in practically all programs with destructive list manipulation.)

We now show how the *Constructive Interpreter* analyzes the above insertion sort program. The top level command is *ci*; it prompts with an asterisk. User input is shown in boldface.

Invoking the debugger, we proceed as follows:

```
| ?- ci.

* qsort(U,V).

Solving goal: qsort([],X) ...

Error detected.   Debugging ...

The goal
    qsort([],[])
is not covered!
```

Since **qsort(U,V)** is symbolic, the debugger first generated a test case *qsort*([ ],*X*) and tried to satisfy it. It discovered that *qsort*([ ],*X*) should have a solution *qsort*([ ],[ ]) according to the specification of *qsort*, but cannot get it from the program we supplied. The debugger therefore reported a bug and tried to fix it.

```
Synthesizing a clause to cover qsort([],[]) ...
Assert clause:
    qsort([],[]) :- true

Listing of qsort(X,Y):
    qsort([],[]) :- true.
    qsort([X|Y],Z) :-
        part(Y,X,W,X1), qsort(W,Z1), qsort(X1,V1),
        append([X|Z1],V1,Z).
```

Since no clause head in the original program unified with *qsort*([ ],[ ]), the debugger used the specification for *qsort* and synthesized the clause

qsort([ ],[ ]) :- *ordered*([ ]),*perm*([ ],[ ])

to cover that goal. Since the body of this clause can be reduced to *true*, the debugger added a unit clause to the program (by asserting it to the database). The goal *qsort*([ ],[ ]) is now satisfiable. Since we initially supplied a symbolic input, we now try for another test case:

```
* Try another test case? y.

Solving goal: qsort([x],X) ...

Error detected.   Debugging ...

The clause
    part([],x,[x],[]) :- true
is false!
```

The debugger now generated a one element list as test input: *qsort*([*x*],*X*). (Note that the input generated, [*x*], contains a skolem constant *x*.) This time, it found an incorrect clause in the procedure *part*, because partitioning an empty list should result in two empty sublist, so the result of *parti*([ ],*x*,*X*,*Y*) should be *part*([ ],*x*,[ ],[ ]) instead of *part*([ ],*x*,[*x*],[ ]). After further analysis, the debugger concludes that:

```
The head of the clause
    part([],X,[X],[]) :- true
is incorrect.   Fixing ...

Cannot fix clause head!
Retract clause:
    part([],X,[X],[]) :- true
Synthesizing a clause to cover part([],x,[],[]) ...
Assert clause:
    part([],X,[],[]) :- true

Listing of part(X,Y,Z,U):
    part([],X,[],[]) :- true.
    part([X|Y],Z,U,[X|W]) :- part(Y,Z,U,W).
    part([X|Y],Z,[X|V],W) :- X <= Z,
                             part(Y,Z,V,W).
```

Since the unit clause in the procedure *part* was incorrect, and the debugger could not fix the head, it retracted the clause. After synthesizing a clause that covers *part*([ ],*x*,[ ],[ ]), the debugger prompts the user:

```
* Try another test case? y.

Solving goal: qsort([0,1],X) ...
Found solution: qsort([0,1],[0,1])
```

The next test case generated is *qsort*([0,1],*X*). This is because we have included the axiom "0 < 1" and a call to the procedure *ordered* (which is part of the specification for *qsort*) causes input variables to be bound to the constants of this axiom. Unlike the previous two test cases, the goal *qsort*([0,1],*X*) is directly solvable by the clauses currently in the program.

```
* Try another test case? y.

Solving goal: qsort([1,0],X) ...

Error detected.   Debugging ...

The clause
    part([0],1,[],[0]) :- part([],1,[],[])
is false!
```

The next test goal *qsort*([1,0],*X*) resulted in the location of an incorrect clause in the procedure *part*. A trace of the procedures shows that the correct solution to *part*([0],1,*X*,*Y*), *viz. part*([0],1,[0],[ ]), can be obtained from the other clause of *part*. Thus, this incorrect clause should have failed, but did not because of a missing subgoal. Our debugger is able to deduce this missing subgoal:

```
There are missing subgoals in the clause:
    part([X|Y],Z,U,[X|W]) :- part(Y,Z,U,W)
Retract erroneous clause:
    part([X|Y],Z,U,[X|W]) :- part(Y,Z,U,W)

Generating missing subgoals ...

Assert clause:
    part([X|Y],Z,U,[X|W]) :- Z <= X,
                             part(Y,Z,U,W)
Listing of part(X,Y,Z,U):
    part([X|Y],Z,U,[X|W]) :- Z <= X,
                             part(Y,Z,U,W).
    part([],X,[],[]) :- true.
    part([X|Y],Z,[X|V],W) :- X <= Z,
                             part(Y,Z,V,W).
```

This time, after correcting for the missing subgoal (by retracting an incorrect clause and asserting a correct one), the debugger reexecuted all the test goals generated so far to make sure the changes do not destroy anything. (Note that there is no way a correctly synthesized clause can cause a problem; retracting an incorrect clause, however, could conceivably cause some goals to become uncovered.)

```
Checking previous goal qsort([],X) ...
Found solution: qsort([],[])

Checking previous goal qsort([x],X) ...
Found solution: qsort([x],[x])

Checking previous goal qsort([0,1],X) ...
Found solution: qsort([0,1],[0,1])

Checking previous goal qsort([1,0],X) ...

Error detected.   Debugging ...

The clause
    qsort([1,0],[1,0]) :-
        part([0],1,[0],[]), qsort([0],[0]),
        qsort([],[]), append([1,0],[],[1,0])
is false!
```

As shown above, the debugger caught another bug when trying to resatisfy the current test goal. Further diagnosis narrows down the bug's location:

```
The clause
    qsort([X|Y],Z) :-
        part(Y,X,W,X1), qsort(W,Z1),
        qsort(X1,V1), append([X|Z1],V1,Z)
contains incorrect subgoals.   Fixing ...

Subgoal "append([X|Y],Z,U)" in clause
    qsort([X|W],U) :-
        part(W,X,U1,V1), qsort(U1,Y),
        qsort(V1,Z), append([X|Y],Z,U)
is incorrect

Trying a local fix ...

Retract clause:
    qsort([X|Y],Z) :-
        part(Y,X,W,X1), qsort(W,Z1),
        qsort(X1,V1), append([X|Z1],V1,Z)
```

```
Assert clause:
    qsort([X|Y],Z) :-
        part(Y,X,W,X1), qsort(W,Z1),
        qsort(X1,V1), append(Z1,[X|V1],Z)


Listing of qsort(X,Y):
    qsort([X|Y],Z) :-
        part(Y,X,W,X1), qsort(W,Z1),
        qsort(X1,V1), append(Z1,[X|V1],Z).
    qsort([],[]) :- true.
```

Up to this point, all the bugs in the original program have been detected and corrected. If we now continue to debug the program, the debugger will keep on generating arbitrarily long lists as test input without reporting an error. We would be led to believe, in this case, that the program is correct with respect to its specifications. (Formal verification of its correctness would require greater theorem proving capabilities.)

We now present a merge sort program to demonstrate how the debugger deals with a looping error.

A Buggy Merge Sort Program

```
                msort([ ],[ ])
                msort(X,Z)      :-  length(X,L),
                                    L1 is L//2,
                                    break(X,L1,X1,X2),
                                    msort(X1,Z1),
                                    msort(X2,Z2),
                                    merge(Z1,Z2,Z)

            break(X,0,[ ],X)
        break([A|X],L,[A|Y],Z)  :-  L1 is L-1,
                                    break(X,L1,Y,Z)

            merge([ ],X,X)
            merge(X,[ ],X)
    merge([A|X],[B|Y],[A|Z])    :-  A ≤ B,
                                    merge(X,[B|Y],Z)
    merge([A|X],[B|Y],[B|Z])    :-  A > B,
                                    merge([A|X],Y,Z)
```

Specifications for the Merge Sort Program		
spec (*msort* (*X* ,*Y*))	:-	*ordered* (*Y*), *perm* (*X* ,*Y*)
spec (*break* (*X* ,*N* ,*Y* ,*Z*))	:-	*append* (*Y* ,*Z* ,*X*), *length* (*Y* ,*N*)
spec (*merge* (*X* ,*Y* ,*Z*))	:-	*rm_list* (*X* ,*Z* ,*Y*), *ordered* (*Z*)
wf o (*msort* (*X* ,*Y*), *msort* (*U* ,*V*))	:-	*shorter* (*X* ,*U*)
wf o (*break* (*X* ,*A* ,*B* ,*C*), *break* (*Y* ,*D* ,*E* ,*F*))	:-	*shorter* (*X* ,*Y*)
wf o (*merge* (*X* ,*Y* ,*A*), *merge* (*U* ,*V* ,*B*))	:-	*shorter* (*X* ,*U*)
wf o (*merge* (*X* ,*Y* ,*A*), *merge* (*U* ,*V* ,*B*))	:-	*shorter* (*Y* ,*V*)

Merge sort accepts a list, breaks it into roughly equivalent halves, recursively merge sorts the sublists, then merges the sorted halves. Note that the predicates used in the above specifications are the same ones used in the specifications for quick sort.

The following is the debugging script:

```
| ?- ci

* msort(U,V).

Solving goal: msort([],X) ...
Found solution: msort([],[])
```

The program has no problem solving the empty list. However, it gets into trouble quickly with the one-element list.

```
* Try another test case? y.

Solving goal: msort([x],X) ...

Error detected.   Debugging ...

The goal  "msort([x],Y)"  in the clause
    msort([x],X) :-
        length([x],1), 0 is 1//2,
        break([x],0,[],[x]), msort([],[]),
        msort([x],Y), merge([],Y,X)
is looping!
```

The debugger found that the procedure call *msort*([*x*],*Y*) to the second clause of *msort* violated the well-founded ordering defined for that recursive procedure.

```
Adding subgoals for well-founded ordering in clause:
    msort(X,Y) :-
        length(X,U), V is U//2,
        break(X,V,Z1,U1), msort(Z1,W1),
        msort(U1,Y2), merge(W1,Y2,Y)

Listing of msort(X,Y):
    msort([],[]):-true.
    msort(X,Y) :-
        shorter(U1,X), length(X,U),
        V is U//2, break(X,V,Z1,U1),
        msort(Z1,W1), msort(U1,Y2),
        merge(W1,Y2,Y)

Continue debugging ...
```

Not being able to fix the offending subgoal directly, the debugger added the well-founded condition as a subgoal to the clause and continues with the analysis.

```
Error detected.   Debugging ...

The goal
    msort([x],[x])
is not covered!

Synthesizing a clause to cover msort([x],[x]) ...
Assert clause:
    msort([X],[X]) :- true

Listing of msort(X,Y):
    msort([X],[X]) :- true.
    msort([],[]) :- true.
    msort(X,Y) :-
        shorter(U1,X), length(X,U),
        V is U//2, break(X,V,Z1,U1),
        msort(Z1,W1), msort(U1,Y2),
        merge(W1,Y2,Y)
```

After enforcing the well-founded ordering on recursive calls, the debugger detected that the program is incomplete, since the goal *msort*([*x*],[*x*]) is not covered. Therefore, it invoked the synthesis routine and found a clause for the goal.

```
Restoring original clause ...
Listing of msort(X,Y):
    msort([X],[X]) :- true.
    msort([],[]) :- true.
    msort(X,Y) :-
        length(X,U), V is U//2,
        break(X,V,Z1,U1), msort(Z1,W1),
        msort(U1,Y2), merge(W1,Y2,Y)
```

Since the error was due to a missing case, the debugger restored the original, correct clause.

It turned out that the looping bug was due to the behavior of procedure *break*. A one-element list to the procedure is always broken into sublists of zero- and one-element. This one-element list is never reduced in the recursive call, and, therefore, need to be treated as a special case. Adding a unit clause for it resolve the problem completely.

Note that, in running the debugger, the user only needs to supply top level goals (in our examples, **qsort(U,V)** and **msort(U,V)**), and types in a *yes* answer for the debugger to continue debugging with alternative test cases. Since the knowledge necessary for the discovery, location, and correction of bugs is either built into the debugger or furnished as program specifications, user intervention during a debugging session is reduced to a minimal level.

8. AUTOMATIC SYNTHESIS OF LOGIC PROGRAMS

There are two main approaches in synthesizing logic programs: deductive and inductive. The next two sections present the main ideas. We then show how these two approaches can be combined to produce a more powerful system, as exemplified by the *Constructive Interpreter*.

8.1. Deductive Synthesis

The deductive synthesis of logic programs can be summarized as follows: formulating intuitively correct *if-and-only-if* logic definitions of the relations of interest, together perhaps with additional facts about those relations, then using this knowledge as an axiom set to derive the procedures by logical deduction [Hogger-81].

The derivation starts with a goal representing the desirable logic procedure and proceeds by applying repeatedly inference rules, until the original goal becomes a set of atomic formula. The first kind of inference rules is for goal simplifications which are meant to modified a goal to Horn clause form by deleting superfluous calls or rewriting complicated calls as conjunctions of simpler ones. The second kind of inference rules is for goal substitutions in which some formula in the axiom set is selected and is substituted for some subformula of a goal. Both of these inference rules might result in bindings of terms to goal

variables.

The procedure derivations are essentially symbolic executions whose procedure invocations and termination criteria are generalizations of their counterparts in normal program executions. In logic programming, a successful computation is obtained by deriving an empty goal, while a successful procedure derivation is obtained by deriving a goal in Horn clause form which is generally nonempty. In the former case, derivation is computing a solution, but in the latter case, derivation is inferring a way of solving the problem.

8.2. Inductive Synthesis

The typical inductive synthesis of logic programs (cf., [Shapiro-83]) starts with a set of examples (in this case, the examples are *solved* goals) which represent the desired behavior of the program to be synthesized. Usually the basic structure of the program and the types of input/output variables are also supplied. For each example (goal), the synthesis system simply searches the set of possible clauses until it finds one which covers the goal in question. The root of the search tree is the predicate of the goal with each argument set to a simple variable and no subgoals. Its descendents are clauses produced by applying simple refinements. If necessary, more clauses can be produced by refining the descendents.

The refinement operator to produce descendents of nodes in the search tree is based on two types of operations: (1) instantiate a variable and (2) add a subgoal to the body of the clause. Instantiation operations instantiate a variable either to a constant, another variable, or a compound term. Output variables are only unified with input variables by means of the *close clause* operation. This sets each of the output variables in a clause to one of the input variables. Clauses produced by this operation are the only ones in which output values will be set.

The inductive synthesis process is basically a generate (refine) and test (diagnose) cycle. The refinement searches for a candidate clause, while the diagnosis validates a clause. It stops when the synthesized program is able to solve all the examples successfully. When diagnosing the correctness of a clause, Shapiro's system often needs to consult an oracle (usually the user) to verify some intermediate results. As will be seen, this oracle can be adequately replaced by executable specifications.

8.3. Program Synthesis with Executable Specifications

The typical process of debugging — designing a test case, detecting an error in the program, locating the error, and fixing it — can be applied in program synthesis. Equipped with procedures for goal generation and checking, the *Constructive Interpreter* can be used to synthesis programs from executable specifications. The following is an example of generating an insertion sort program, with very little user involvement. In fact, the user only needs to type in the initial request, and answer "yes" or "no" when the system prompts for instruction on whether to continue generate new goals. This example starts with an empty

isort program, however, the specifications of *isort* mentioned in section 5, that is,

Specifications for the Insertion Sort Program		
spec (*isort* (*X* ,*Y*))	:-	*ordered* (*Y*), *perm* (*X* ,*Y*)
spec (*insert* (*X* ,*Y* ,*Z*))	:-	*ordered* (*Y*) → *ordered* (*Z*), *perm* ([*X* |*Y*],*Z*)

is given to the system.

```
| ?- ci.

* isort(U,V).
Solving goal: isort([],X) ...
Error: missing solution isort([],[]).
    diagnosing...
Error diagnosed: isort([],[]) is uncovered.
```

After starting, the system first generated a goal *isort* ([ ],*X*) which is of course not covered, since the initial program is empty. Therefore the system searches for a clause to cover this goal.

```
Searching for a cover to isort([],[])...
Checking: isort(X,X):-true
Found clause: isort(X,X):-true
    after searching 4 clauses.

Listing of isort(X,Y):
    isort(X,X):-true.
```

With little effort, the system easily found a unit clause that does the job. The program synthesized so far contains exactly one clause:

isort (*X* ,*X*) :- *true* .

Now that the goal *isort* ([ ],*X*) is solved, the system prompts for continuation.

```
* Continue? y.
Solving goal: isort([x],Y) ...
Found solution: isort([x],[x]).

Checking fact(s)...no error found.
```

```
* Continue? y.
Solving goal: isort([0,1],X) ...
Found solution: isort([0,1],[0,1]).

Checking fact(s)...no error found.
```

The current clause solved two successive input list [*x*] and [0,1] without any problem. However, it cannot handle the next generated input list [1,0], shown as follows:

```
* Continue? y.
Solving goal: isort([1,0],X) ...
Error: wrong solution isort([1,0],[1,0]).
    diagnosing...
Error diagnosed: isort([1,0],[1,0]):-true is false.

Listing of isort(X,Y):

Checking fact(s)...
Error: missing solution isort([],[]).
    diagnosing...
Error diagnosed: isort([],[]) is uncovered.
```

The system found that the clause

isort(*X*,*X*) :- *true*

is too general, since it just simply returns an output list which is exactly the same as the input list. The debugger removed this clause. However, it soon found out that it can no longer solved the first goal *isort*([ ],[ ]), since the program is empty now.

```
Searching for a cover to isort([],[])...
Checking: isort(X,[]):-true
Found clause: isort(X,[]):-true
    after searching 4 clauses.

Listing of isort(X,Y):
    isort(X,[]):-true.
```

After some search, the system found a clause that returns an empty list as output, regardless of input. This clause covers the goal *isort*([ ],[ ]), however, it is incorrect for any goal of *isort* with a non-empty list of input.

```
Checking fact(s)...
Error: missing solution isort([x],[x]).
    diagnosing...
Error diagnosed: isort([x],[x]) is uncovered.

Searching for a cover to isort([x],[x])...
```

```
Checking: isort([X|Y],Z):-insert(X,Y,Z)
Found clause: isort([X|Y],Z):-insert(X,Y,Z)
    after searching 20 clauses.

Listing of isort(X,Y):
    isort(X,[]):-true.
    isort([X|Y],Z):-insert(X,Y,Z).
```

The system found a clause that could cover the goal *isort*([*x*],[*x*]). However, there is no clause in the program that can cover the subgoal *insert*(*X*,*Y*,*Z*) in the newly found clause. Therefore, the system continues the search:

```
Checking fact(s)...
Error: missing solution isort([x],[x]).
    diagnosing...
Error diagnosed: insert(x,[],[x]) is uncovered.

Searching for a cover to insert(x,[],[x])...
Checking: insert(X,Y,[X|Y]):-true
Found clause: insert(X,Y,[X|Y]):-true
    after searching 6 clauses.

Listing of insert(X,Y,Z):
    insert(X,Y,[X|Y]):-true.
```

After adding a clause for the procedure *insert*, the program is still not complete.

```
Checking fact(s)...
Error: missing solution isort([1,0],[0,1]).
    diagnosing...

Error diagnosed: insert(1,[0],[0,1]) is uncovered.

Searching for a cover to insert(1,[0],[0,1])...

Checking: insert(X,[Y|Z],[Y,X|Z]):-true
Found clause: insert(X,[Y|Z],[Y,X|Z]):-true
    after searching 27 clauses.

Listing of insert(X,Y,Z):
    insert(X,Y,[X|Y]):-true.
    insert(X,[Y|Z],[Y,X|Z]):-true.
```

The system added yet another clause for *insert*, but upon rechecking, *isort* still returns an incorrect result for input [1,0].

```
Checking fact(s)...
Error: wrong solution isort([1,0],[1,0]).
    diagnosing...

Error diagnosed: insert(1,[0],[1,0]):-true is false.

Listing of insert(X,Y,Z):
    insert(X,[Y|Z],[Y,X|Z]):-true.
```

This time, the first clause in procedure *insert* is identified as false, since according to the specification of *insert*, inserting the element 1 into the list [0] should result in the output [0,1] instead of [1,0] as computed by the erroneous clause. Removing the clause, the system is in search of a replacement.

```
Checking fact(s)...
Error: missing solution isort([x],[x]).
    diagnosing...
Error diagnosed: insert(x,[],[x]) is uncovered.

Searching for a cover to insert(x,[],[x])...
Checking: insert(X,Y,[X]):-true
Found clause: insert(X,Y,[X]):-true
    after searching 9 clauses.

Listing of insert(X,Y,Z):
    insert(X,[Y|Z],[Y,X|Z]):-true.
    insert(X,Y,[X]):-true.
```

The new clause does not solve the problem, as the system still complains about errors:

```
Checking fact(s)...
Error: missing solution isort([0,1],[0,1]).
    diagnosing...

Error diagnosed: insert(0,[1],[0,1]) is uncovered.

Searching for a cover to insert(0,[1],[0,1])...

Checking: insert(X,[Y|Z],[X|V]):-insert(Y,Z,V)
Refuted: insert(1,[0],[1,0]):-insert(0,[],[0])
```

Another new clause was found but was immediately rejected, since it does not compute the correct result.

```
Checking: insert(X,[Y|Z],[X,Y|Z]):-X<Y
Found clause: insert(X,[Y|Z],[X,Y|Z]):-X<Y
```

```
    after searching 33 clauses.

Listing of insert(X,Y,Z):
    insert(X,[Y|Z],[Y,X|Z]):-true.
    insert(X,Y,[X]):-true.
    insert(X,[Y|Z],[X,Y|Z]):-X<Y.
```

A clause with an arithmetic test as a subgoal is now added to the program. The program is now correct with respect to all the known *facts* to the system. (A *fact* is defined as a ground term with a value of *true* or *false*; it may be supplied by the user, or generated by the system when solving goals. For example, <*isort*([1,0],[0,1]),*true*> is a fact, so is <*isort*([1,0],[1,0]),*false*>. A correct program should succeed on a *true fact*, and fail on a *false fact*.) The system proceeds to check if the program satisfies the goals generated so far:

```
Checking fact(s)...no error found.
Checking previous goals ...
The goal: isort([],[]) succeeded.

Error: wrong solution isort([x],[]).
    diagnosing...
Error diagnosed: isort([x],[]):-true is false.

Listing of isort(X,Y):
    isort([X|Y],Z):-insert(X,Y,Z).
```

Quickly, a false clause was found and removed. Expectedly, this creates problems.

```
Checking fact(s)...
Error: missing solution isort([],[]).
    diagnosing...
Error diagnosed: isort([],[]) is uncovered.

Searching for a cover to isort([],[])...
Checking: isort([],[]):-true
Found clause: isort([],[]):-true
    after searching 5 clauses.

Listing of isort(X,Y):
    isort([X|Y],Z):-insert(X,Y,Z).
    isort([],[]):-true.
```

The search found a clause to cover the case when the input list is empty. When trying to recompute the goal *isort*([0,1],*X*), another bug is found.

```
Checking fact(s)...no error found.
Checking previous goals ...
```

```
The goal: isort([],[]) succeeded.
The goal: isort([x],[x]) succeeded.

Error: wrong solution isort([0,1],[1,0]).
    diagnosing...

Error diagnosed: insert(0,[1],[1,0]):-true is false.

Listing of insert(X,Y,Z):
    insert(X,Y,[X]):-true.
    insert(X,[Y|Z],[X,Y|Z]):-X<Y.
```

The system tracked down a false clause in procedure *insert* and continues the search:

```
Checking fact(s)...
Error: missing solution isort([1,0],[0,1]).
    diagnosing...
Error diagnosed: insert(1,[0],[0,1]) is uncovered.

Searching for a cover to insert(1,[0],[0,1])...
Checking: insert(X,[Y|Z],[Y|V]):-insert(X,Z,V)
Refuted: insert(0,[1],[1,0]):-insert(0,[],[0])
Checking: insert(X,[Y|Z],[Y,X|Z]):-Y<X
Found clause: insert(X,[Y|Z],[Y,X|Z]):-Y<X
    after searching 46 clauses.

Listing of insert(X,Y,Z):
    insert(X,Y,[X]):-true.
    insert(X,[Y|Z],[X,Y|Z]):-X<Y.
    insert(X,[Y|Z],[Y,X|Z]):-Y<X.
```

After some extensive search, the system found a clause which complements another clause with similar structure, but opposite arithmetic test in procedure *insert*. However, the clause for the base case in *insert* is still incorrect.

```
Checking fact(s)...no error found.
Checking previous goals ...
The goal: isort([],[]) succeeded.
The goal: isort([x],[x]) succeeded.

Error: wrong solution isort([0,1],[0]).
    diagnosing...

Error diagnosed: insert(0,[1],[0]):-true is false.

Listing of insert(X,Y,Z):
```

```
insert(X,[Y|Z],[X,Y|Z]):-X<Y.
insert(X,[Y|Z],[Y,X|Z]):-Y<X.
```

The false clause is now to be replaced.

```
Checking fact(s)...
Error: missing solution isort([x],[x]).
    diagnosing...
Error diagnosed: insert(x,[],[x]) is uncovered.

Searching for a cover to insert(x,[],[x])...
Checking: insert(x,[],[x]):-true
Found clause: insert(x,[],[x]):-true
    after searching 11 clauses.

Listing of insert(X,Y,Z):
    insert(X,[Y|Z],[X,Y|Z]):-X<Y.
    insert(X,[Y|Z],[Y,X|Z]):-Y<X.
    insert(X,[],[X]):-true.
```

Up to this point, the program synthesized solves the generated goals successfully, as follows:

```
Checking fact(s)...no error found.
Checking previous goals ...
The goal: isort([],[]) succeeded.
The goal: isort([x],[x]) succeeded.
The goal: isort([0,1],[0,1]) succeeded.
The goal: isort([1,0],[0,1]) succeeded.
```

We now try the program on input lists of three elements.

```
* Continue? y.
Solving goal: isort([0,1,2],X) ...

Found solution: isort([0,1,2],[0,1,2]).
Checking fact(s)...no error found.
Checking previous goals ...
The goal: isort([],[]) succeeded.
The goal: isort([x],[x]) succeeded.
The goal: isort([0,1],[0,1]) succeeded.
The goal: isort([1,0],[0,1]) succeeded.
The goal: isort([0,1,2],[0,1,2]) succeeded.
```

It did not encounter any problems when the input list is sorted. However, a bug in procedure *isort* is revealed by the next test case.

```
* Continue? y.
Solving goal: isort([0,2,1],X) ...
Error: wrong solution isort([0,2,1],[0,2,1]).
    diagnosing...

Error diagnosed:
    isort([0,2,1],[0,2,1]):-insert(0,[2,1],[0,2,1])
is false.

Listing of isort(X,Y):
    isort([],[]):-true.
```

Removing the offending clause, the system starts a new search.

```
Checking fact(s)...
Error: missing solution isort([x],[x]).
    diagnosing...
Error diagnosed: isort([x],[x]) is uncovered.

Searching for a cover to isort([x],[x])...

Checking: isort([X|Y],[X|U]):-isort(Y,U)
Refuted: isort([1,0],[1,0]):-isort([0],[0])

Checking: isort([X|Y],Z):-isort(Y,V),insert(X,V,Z)
Found clause: isort([X|Y],Z):-isort(Y,V),insert(X,V,Z)
    after searching 65 clauses.

Listing of isort(X,Y):
    isort([],[]):-true.
    isort([X|Y],Z):-isort(Y,V),insert(X,V,Z).
```

The system finally found the clause for the recursive case of *isort*.

```
Checking fact(s)...no error found.
Checking previous goals ...
The goal: isort([],[]) succeeded.
The goal: isort([x],[x]) succeeded.
The goal: isort([0,1],[0,1]) succeeded.
The goal: isort([1,0],[0,1]) succeeded.
The goal: isort([0,1,2],[0,1,2]) succeeded.
The goal: isort([0,2,1],[0,1,2]) succeeded.
```

We now try a new test case.

```
* Continue? y.
Solving goal: isort([1,0,2],X) ...
Found solution: isort([1,0,2],[0,1,2]).
```

Since no error was found, there was no need to recheck all the facts and goals. We continue as follows:

```
* Continue? y.
Solving goal: isort([1,2,0],X) ...
Found solution: isort([1,2,0],[0,1,2]).
```

Just when everything seems all right, another bug is revealed, as follows:

```
* Continue? y.
Solving goal: isort([2,0,1],X) ...
Error: wrong solution isort([2,0,1],[0,2,1]).
    diagnosing...

Error diagnosed: insert(2,[0,1],[0,2,1]):-0<2 is false.

Listing of insert(X,Y,Z):
    insert(X,[Y|Z],[X,Y|Z]):-X<Y.
    insert(X,[],[X]):-true.
```

The procedure of *insert* is still incorrect.

```
Checking fact(s)...
Error: missing solution isort([1,0],[0,1]).
    diagnosing...
Error diagnosed: insert(1,[0],[0,1]) is uncovered.

Searching for a cover to insert(1,[0],[0,1])...
Checking: insert(X,[Y|Z],[Y,X|Z]):-Y=<X
Refuted: insert(2,[0,1],[0,2,1]):-0=<2

Checking: insert(X,[Y|Z],[Y|V]):-insert(X,Z,V),Y<X
Found clause: insert(X,[Y|Z],[Y|V]):-insert(X,Z,V),Y<X
    after searching 53 clauses.

Listing of insert(X,Y,Z):
    insert(X,[Y|Z],[X,Y|Z]):-X<Y.
    insert(X,[],[X]):-true.
    insert(X,[Y|Z],[Y|V]):-insert(X,Z,V),Y<X.
```

Finally, a clause for the recursive case of *insert* is found.

```
Checking fact(s)...no error found.
Checking previous goals ...
The goal: isort([],[]) succeeded.
The goal: isort([x],[x]) succeeded.
The goal: isort([0,1],[0,1]) succeeded.
The goal: isort([1,0],[0,1]) succeeded.
The goal: isort([0,1,2],[0,1,2]) succeeded.
The goal: isort([0,2,1],[0,1,2]) succeeded.
The goal: isort([1,0,2],[0,1,2]) succeeded.
The goal: isort([1,2,0],[0,1,2]) succeeded.
The goal: isort([2,0,1],[0,1,2]) succeeded.
```

The last permutation of the three-element list now executes correctly on the synthesized program.

```
* Continue? y.
Solving goal: isort([2,1,0],X) ...
Found solution: isort([2,1,0],[0,1,2]).

* Continue? n.

* exit.

yes
```

We stop here with the synthesized program as follows:

A Synthesized Program of Insertion Sort		
isort ([ ],[ ])		
isort ([*X* | *Y*],*Z*)	:-	*isort* (*Y* ,*V*), *insert* (*X* ,*V* ,*Z*)
insert (*X* ,[*Y* | *Z*],[*X* ,*Y* | *Z*])	:-	*X* < *Y*
insert (*X* ,[ ],[*X*])		
insert (*X* ,[*Y* | *Z*],[*Y* | *V*])	:-	*insert* (*X* ,*Z* ,*V*), *Y* < *X*

9. CONCLUSION

In this research, we have tried to utilize user-supplied program specifications, as much as possible, for debugging purposes. We have designed a methodology which uses executable specifications to generate test cases, to locate and fix program errors, and the combination of these functions can also be used to synthesize programs automatically. In this section, we discuss the strengths and weaknesses of our approach, compare it with related work, and consider directions for future research.

9.1. Testing, Debugging, and Program Proving

The traditional testing approach is only concerned with designing test cases that might show a program to be incorrect (cf. [Dijkstra-76]). Although there have been attempts to show that thorough testing can sometimes actually guarantee the correctness of a program (cf. [Goodenough-Gerhart-75, Brooks-80], this has not been the case in actual software practice. Regardless, there is a fundamental problem that is not addressed: just testing a program and trying to detect errors will not make the program correct. One still has to fix the errors detected. In other words, testing alone will not be sufficient for generating a correct program, since knowing that a program is incorrect does not imply knowing the cause, and certainly not the fix. Research in testing provides, at most, methods to disclose the existence of bugs in a program, but does not deal directly with the problem of locating and correcting bugs.

Most of the work on debugging, on the other hand, emphasizes the classification of errors, strategies of locating errors, and methods of repairing errors. They rely on test cases generated using some other means to show first that the program is incorrect, before their debugging techniques can be applied. For example, to use Shapiro's system ([Shapiro-83]), one has to know beforehand what kind of errors one is dealing with (or one can supply such information to the debugger by answering a number of queries generated by the debugger), then invoke different routines for different kind of errors. Calling the procedure that deals with missing clauses when the symptom is due to an incorrect clause will not solve the problem.

Recent research and technical development in program proving have led some to hope that formal correctness proofs of programs can replace testing and debugging as a means for insuring that programs will perform as what they are prescribed to do. However, as have been argued by others (cf, [DeMillo,*etal.*-79, Gerhart-Yelowitz-76]) this may not be possible. Due to human fallibility in the practice of mathematics, it is better to apply as many techniques as possible when trying to show that a program is correct, since different methodologies often have complementary strengths and weaknesses.

Our methodology is intended to combine the functions of testing, debugging, and proving under one uniform framework. We are able to test a program using sample data, debug it if errors are detected, and finally apply proof techniques to show that the debugged program is consistent with its specifications. We have demonstrated that this approach is feasible with logic programming.

9.2. Executable Specifications

Formal specifications can facilitate the software development process. Besides serving as a guide in programming, they allow for formal reasoning about correctness. Being a special case of formal specifications, executable specifications allow for rapid prototyping and computer-aided debugging. In other words, executable specifications can be considered as a prototype of the system being built, and can be used to mechanically evaluate the desired functionalities of the target system.

In our work, we have used the logic programming language, Prolog, to specify Prolog programs. These specifications are actually high-level, nonalgorithmic programs. When they contain a "generator" predicate and axioms for system predicates, they can be used to automatically generate test cases by solving input-output pairs. Due to the simplicity of data in Prolog (only list structures are available), errors are usually uncovered at an early stage in the debugging cycle, before the test inputs get very complicated. Thus, this method provides sufficient test data for typical debugging needs.

A major use of executable specifications in our work is for automated detection and location of errors in programs. When a debugger, be it a human or a machine, is to debug a program, it must know the intended behavior of the program. That is, it must know *a priori* what to expect for the result of running the program. Executable specifications provide not only such vital information about the program, but also in a form that can be processed by the computer directly. We use executable specifications of each procedure to check intermediate results of program execution. Any discrepancy between the actual and expected outcome of a procedure execution can be detected as soon as it is encountered, and the location where the error occurs can also be determined (at the clause level).

After an error is detected and located, specifications can be used to help fix the error. When the error is due to an incorrect clause, a deductive proof will be constructed in an attempt to repair the erroneous clause by adding additional subgoals to the clause or fixing an incorrect subgoal. During the proof process, the specifications are used to replace goals (or subgoals) in the clause so the goals (or subgoals) can be reduced to simpler or primitive forms to complete the proof. If the error is due to a missing clause, an inductive inference routine will be invoked to synthesize a clause to solve an uncovered goal. In this case, the specifications provide information for pruning the search tree and for guiding the search for a correct clause.

As summarized above, the use of executable specifications in our system automates many of the difficult tasks during the program debugging process.

9.3. Deductive vs. Inductive Reasoning

Deduction and induction are very different inference mechanisms. In this work, we have shown that they can complement each other in the debugging process.

Logical deduction is a powerful technique in the sense that the result from deductive

inference is guaranteed correct (or consistent with the axioms). In the context of logic programming, deduction can be used to execute, derive, transform, or verify programs. We have applied this procedure to check for inconsistency between a program and its specifications. Also, in the process of verifying a clause, deduction is used to help identify additional subgoals that suffice to make the clause correct.

Inductive inference refers to the process of discovering general rules from scattered facts. When applied in logic programming, this technique is used to generate program clauses from a set of known goals. We use induction to suggest program clauses from the generated input-output pairs, whenever a goal is detected that should be covered by the program, but is not (i.e., when incompleteness is identified).

Our debugging system for Prolog programs combines both deductive and inductive routines. Deduction is used to locate errors and fix clauses; induction is employed to find missing clauses. The system can automatically invoke either procedures as required by the state of the program being diagnosed.

9.4. Comparison with Related Work

Like [Shapiro-83], our work is concerned with debugging logic programs. Unlike [Shapiro-83], we use executable specification as the basis for judging whether a program statement is correct, rather than querying the user about the correctness of particular instances of statements. On account of this feature, our system is able to debug a program with minimum ongoing user intervention. This approach also avoids the likelihood that a user may not be able to provide the required information or may unintentionly give wrong answers to queries generated. On the other hand, we must presume that the user-supplied specifications accurately reflect the program requirements.

Since we are able to use specifications to generate test cases and then execute the test data directly, the bug location routines for different kinds of errors have been integrated in one diagnostic interpreter. We do not need to find out what kind of error is detected before calling the diagnostic procedure for that specific kind of error. In other words, our implementation can be used as a regular program interpreter, and it can be used to compute output results just like any standard Prolog interpreter. When supplied with the specifications of a program, however, all the computations are monitored by the system and any results inconsistent with the specifications will invoke the debugging routines to locate and fix the errors.

Another major difference between our system and Shapiro's is that our system performs a detailed bug-analysis whenever an error is encounted, and different bug correction strategies are used depending on the nature and context of error. In Shapiro's system, whenever a clause is found incorrect, it is removed from the program and a new clause is generated, by searching through a class of equivalent clauses that supposedly cover the goals that have become uncovered by removal of the offending clause. However, since our bug analysis procedure is more finely tuned than Shapiro's, it always tries to first fix the error-laden clause as much as possible before disposing of it and synthesizing a new one.

We believe this is a more economical and efficient strategy, since most programs should be fairly close to being correct (cf., [Budd, *et.al.*-80]).

Our work is also different from the Programmer's Apprentice (PA) project [Waters-85] which deals with knowledge-based program synthesis. In PA, the human acts as the "chief programmer", responsible for making all the difficult, important decisions regarding program construction, while the system is the assistant in charge of providing lower-level language support. The backbone of the PA approach is a collection of *programming cliches* (abstract version of programming plans) which provide domain specific algorithms to help the programmer construct programs. The usefulness of the PA system for solving a particular problem, therefore, depends on whether the cliches necessary for the program at hand are available. The number of important cliches that a versatile system requires is usually several thousand. Our system does not rely on such huge number of cliches for proper functioning. Instead, it uses executable specifications and general reasoning strategies (deduction and induction) for program analysis.

9.5. Issues to be Addressed in Future Research

In our work, we have used specifications as the criterion for judging correctness. In other words, we are assuming the specifications themselves to be error-free. The debugging system will obviously not perform properly, given erroneous specifications. In actual software practice, when there is something wrong with the specifications for a program, the process of programming and verification will be fallacious anyway. The debugging of specifications is an important topic, but is beyond the scope of our research.

Another problem occurs when the program is under-specified. Since our bug location routine needs information about each procedure to determine if a procedure is producing the correct result, we need to supply lower-level specifications to the debugging system. As has been argued before, an automatic debugging system must know the expected behavior of the program or procedure being debugged. This information has to be supplied along with top-level specifications or generated by some other means (such as querying the programmer during a debugging session). In our implementation, whenever a procedure's specifications is not present, our system just assumes that the procedure is correct and there is no need to debug it.

In specifying the intended behavior of a program, we have used Prolog, taking advantage of its executability. However, since Prolog only implements a limited subset of first order predicate logic (which is a prevailing language for writing formal specifications), its power and expressiveness as a specification language is limited. It may be more desirable to use more expressive languages such as those mentioned in section 4 or full predicate calculus. A primary task would then be to build a mechanical system that could process such specifications automatically and that can be integrated in an automated program development environment. A related issue would be the verifications of specifications. I.e., we need a means to check the correctness of specifications.

Though our system currently uses generator predicates to produce test data, it may be

more efficient and effective if a testing system employs specific techniques (e.g., data flow analysis [Rapps-Weyuker-85]) such that all errors can be detected with the fewest number of test runs. In other words, we want to make the test cases more specific to the problem being tackled.

The bug correction process is the most difficult step for an automated debugging system. This is because it is difficult to formalize the complex knowledge involved in reasoning about the problem to be solved, the algorithm used, the program that has been written, the input/output specifications, and the semantics of the program language. In our system, we have relied on the specifications of programs and their procedures, the operational semantics of Prolog, a deductive proof mechanism, and an inductive routine. As discussed in section 6, the deductive mechanism is equipped with proof rules, while the inductive routine is augmented with some heuristics. The proof rules use specifications and program clauses to reduce goals. Although these rules are in general applicable to all cases, due to the limitations in the language (i.e., Prolog) in which the prover is implemented, a certain sequential order is observed during the proof process. Therefore, one may obtain a different result if the order of proof rules is changed. The heuristics in the inductive routine are ad-hoc in nature. We believe our system can benefit from the addition of a knowledge base consisting of both specific heuristics concerning program debugging and algorithmic information concerning the solutions to particular problems.

In conclusion, in this research we have demonstrated that, in the realm of logic programming, the tedious problem of program debugging is perhaps amenable to automation.

REFERENCES

A. Adam and J. P. Laurent. "LAURA, a system to debug student programs," *Artificial Intelligence* (1980), Vol. 15, pp. 75-122.

K. R. Apt and M. H. v. Emden. "Contributions to the theory of logic programming," *J. of the Association for Computing Machinery* (July 1982), Vol. 29, pp. 841-862.

J. Backus. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Communications ACM* (1978), Vol. 21, pp. 613-641.

G. F. Badger Jr., R. H. Campbell, N. Dershowitz, M. T. Harandi, A. L. Laursen, R. S. Michalski, D. Michie, R. Penka and M. Simmonds. "Knowledge based programming assistant", Report DCS-F-82-894, Department of Computer Science, University of Illinois, Urbana, IL, April 1982.

D. Barstow, R. Duffey, S. Smoliar and S. Vestal. "An automatic programming system to support an experimental science," *Sixth International Conference on Software Engineering* (September 1982), pp. 360-366.

B. W. Boehm. "Software Engineering," *IEEE Transactions on Computers* (December 1976), Vol. C-25, pp. 1226-1241.

M. Brooks. "Determining Correctness by Testing", Stanford Artificial Intelligence Laboratory, May 1980.

M. Bruynooghe and L. M. Pereira. "Deduction revision through intelligent backtracking," In: *Issues in Prolog Implementation*, J. Campbell, ed. Ellis Horwood Ltd., 1984.

T. Budd, R. DeMillo, R. Lipton and F. Sayward. "Theoretical and empirical studies on using program mutation to test the functional correctness of programs," *Proceedings of the Seventh ACM Symposium on Principles of programming languages* (January 1980).

K. L. Clark. "The synthesis and verification of logic programs", Research Report DOC 81/36, Department of Computing, Imperial College, London, England, September 1981.

L. A. Clarke. "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering* (September 1976), Vol. SE-2, pp. 215-222.

L. A. Clarke and D. J. Richardson. "Symbolic evaluation methods for program analysis," In: *Program Flow Analysis: Theory and Applications*, S. S. Muchnick, ed. Prentice-Hall, Inc., Englewood Cliffs, N. J., 1981, pp. 265-300.

W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer, New York, 1984.

D. Cohen, W. Swartout and R. Balzer. "Using symbolic execution to characterize behavior," *ACM Software Enginnering Notes* (December 1982), Vol. 7, No. 5, pp. 25-32.

J. Darlington, A. J. Field and H. Pull. "The unification of functional and logic languages," In: *Logic Programming: Relations, Functions, and Equations*, D. DeGroot and G. Lindstrom, eds. Prentice Hall, Englewood Cliffs, NJ, 1985.

R. A. DeMillo, R. J. Lipton and A. J. Perlis. "Social processes and proofs of theorems and programs," *Communications of the ACM* (May 1979), Vol. 22, No. 5, pp. 271-280.

N. Dershowitz. *The Evolution of Programs*. Birkhauser, Boston, MA, 1983.

N. Dershowitz. "Equations as programming language," *Proceedings of the Fourth Jerusalem Conference on Information Technology* (May 1984), pp. 114-123.

N. Dershowitz and N. A. Josephson. "An efficient implementation of narrowing: The RITE way," *Proceedings of the Third IEEE Symposium on Logic Programming* (September 1986).

E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

K. Fuchi and K. Furukawa. "The role of logic programming in the fifth generation computer project," *Third International Conference on Logic Programming* (July 1986), pp. 1-24.

S. L. Gerhart and L. Yelowitz. "Observations of fallibility in applications of modern programming methodologies," *IEEE Transactions on Software Engineering* (September 1976), Vol. SE-2, No. 3, pp. 195-207.

J. A. Goguen and J. Meseguer. "Equality, types, modules and (why not?) generics for logic programming," *Logic Programming* (1984), Vol. 1, No. 2, pp. 179-210.

J. A. Goguen and J. J. Tardo. "An introduction to OBJ: A language for writing and testing formal algebraic specifications," *Proceedings of the Specification of Reliable Software Conference* (April 1979), pp. 170-189.

J. B. Goodenough and S. L. Gerhart. "Toward a theory of test data selection," *Sigplan Notices* (June 1975), Vol. 10, No. 6, pp. 493-510.

J. V. Guttag and J. J. Horning. "The algebraic specification of abstract data types," *Acta Informatica* (1978), Vol. 10, No. 1, pp. 27-52.

M. T. Harandi. "Knowledge-based program debugging: A heuristic model," *Proceedings of SOFTFAIR* (1983), pp. 282-288.

C. A. R. Hoare. "An axiomatic basis for computer programming," *Communications of the ACM* (October 1969), Vol. 12, No. 10, pp. 576-583.

C. J. Hogger. "Derivation of logic programs," *J. of the Association for Computing Machinery* (April 1981), Vol. 28, No. 2, pp. 372-392.

W. E. Howden. "Symbolic testing and the DISSECT symbolic evaluation system," *IEEE Transactions on Software Engineering* (July 1977), Vol. SE-3, pp. 266-278.

W. E. Howden. "A functional approach to program testing and analysis," *IEEE Transactions on Software Engineering* (October 1986), Vol. SE-12, No. 10, pp. 997-1005.

W. L. Johnson and E. Soloway. "PROUST: Knowledge-based program understanding," *IEEE Transactions on Software Engineering* (March 1985), Vol. SE-11, No. 3, pp. 267-275.

S. M. Katz and Z. Manna. "Towards automatic debugging of programs," *Proceedings of the International Conference on Reliable Software* (April 1975), pp. 143-155.

S. Katz and Z. Manna. "Logical analysis of programs," *Communications of the ACM* (April 1976), Vol. 19, No. 4, pp. 188-206.

J. C. King. "Symbolic execution and program testing," *Communications of the ACM* (July 1976), Vol. 19, No. 7, pp. 385-391.

H. J. Komorowski and J. Maluszynski. "Logic programming and rapid prototyping", TR-01-86, Harvard University, 1986.

R. A. Kowalski. "Predicate logic as programming language," *Proceedings of the IFIP Congress* (1974), pp. 569-574.

R. A. Kowalski. "Algorithm = logic + control," *Journal of the Association for Computing Machinery* (1979), No. 22, pp. 425-436.

R. A. Kowalski. *Logic for Problem Solving*. North-Holland, Amsterdam, 1979.

R. Kowalski. "The relation between logic programming and logic specification," In: *Mathematical Logic and Programming Languages*, C. A. R. Hoare and Shepherdson, eds. Prentice/Hall International, Inc., Englewood Cliffs, New Jersey, 1985.

R. A. Kowalski and M. H. van Emden. "The semantics of predicate logic as a programming language," *J. of the Association for Computing Machinery* (October 1976), Vol. 23, pp. 733-742.

B. H. Liskov and V. Berzins. "An appraisal of program specifications," In: *Software Specification Techniques*, Narain Gehani, ed. Addison-Wesley, 1986, pp. 3-23.

J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, New York, 1984.

D. W. Loveland. *Automated Theorem Proving: A Logical Basis*. North-Holland, New York, 1978.

J. McCarthy. "Recursive functions of symbolic expressions and their computation by machine, Part I," *Communications of the ACM* (April 1960), Vol. 3, pp. 184-195.

J. R. McGraw. "Data flow computing - software development," *IEEE Transactions on Computers* (December 1980), Vol. C-29, pp. 1095-1103.

E. F. Miller Jr. "Software testing technology: An overview," In: *Handbook of Software Engineering*, C. R. Vick and C. V. Ramamoorthy, eds. Van Nostrand Reinhold Company Inc., New York, 1984, pp. 359-379.

W. R. Murray. "Automatic Program Debugging for Intelligent Tutoring Systems", The University of Texas at Austin, Austin, Texas, June 1986.

G. J. Myers. *Software Reliability: Principles & Practices*. John Wiley & Sons Inc., New York, 1976.

G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., New York, 1979.

L. M. Pereira. "Rational debugging in logic programming," *Proceedings of the Third International Conference on Logic Programming* (July 1986), pp. 203-210.

D. A. Plaisted. "An efficient bug location algorithm," *Proceedings of the Second International Logic Programming Conference* (July, 1984).

S. Rapps and E. J. Weyuker. "Selecting software test data using data flow information," *IEEE Transactions on Software Engineering* (April 1985), Vol. SE-11, pp. 367-375.

S. A. Renner. "Diagnosis of logical errors in Pascal programs", Report UIUC-DCS-F-84-915, University of IL, Urbana, April 1984.

J. A. Robinson. "A machine-oriented logic based on the resolution principle," *J. of the Association for Computing Machinery* (Jan 1965), Vol. 12, No. 1, pp. 23-41.

G. Ruth. "Intelligent program analysis," *Artificial Intelligence* (1976), Vol. 7, pp. 65-85.

E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983.

D. R. Smith. "Derived preconditions and their use in program synthesis," *Proceedings of the Sixth Conference on Automated Deduction* (June 1982), pp. 172-193.

S. W. Smoliar. "Approaches to executable specifications," *ACM Software Enginnering Notes* (December 1982), Vol. 7, No. 5, pp. 155-159.

J. M. Spitzen, K. N. Levitt and L. Robinson. "An example of hierarachical design and proof", Report, August 1977.

R. J. Waldinger and K. N. Levitt. "Reasoning about programs," *Artificial Intelligence* (Fall 1974), Vol. 5, No. 3, pp. 235-316.

R. C. Waters. "The programmer's apprentice: a session with KBEmacs," *IEEE Transactions on Software Engineering* (November 1985), Vol. SE-11, pp. 1296-1320.

Distribution List

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Library, Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Center for Naval Analyses 4401 Ford Ave. Alexandria, VA 22302-0268	1
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Chairman, Computer Science Department Code 52 Naval Postgraduate School Monterey, CA 93943	1
Prof. Yuh-jeng Lee Code 52Le Naval Postgraduate School Monterey, CA 93943	100

DUDLEY KNOX LIBRARY
3 2768 00329685 6